TÉCNICO+
FORMAÇÃO AVANÇADA

Deep Learning

Gonçalo M. Correia

GALP 2023

Program for today

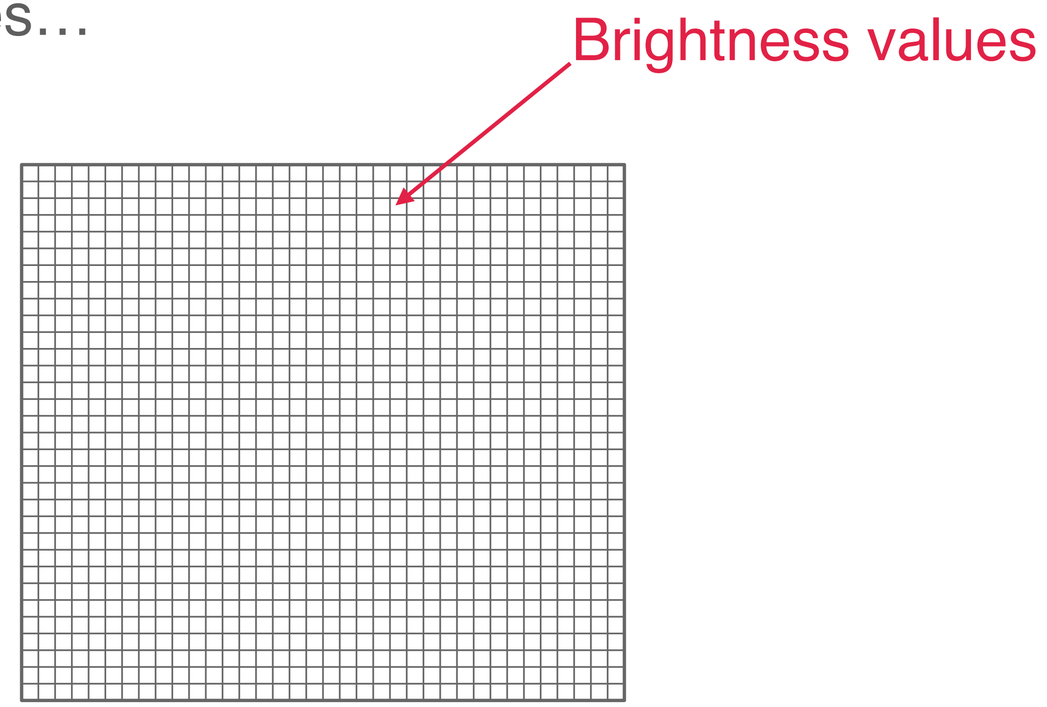
Representations

Recurrent neural networks

GRUs and LSTMs

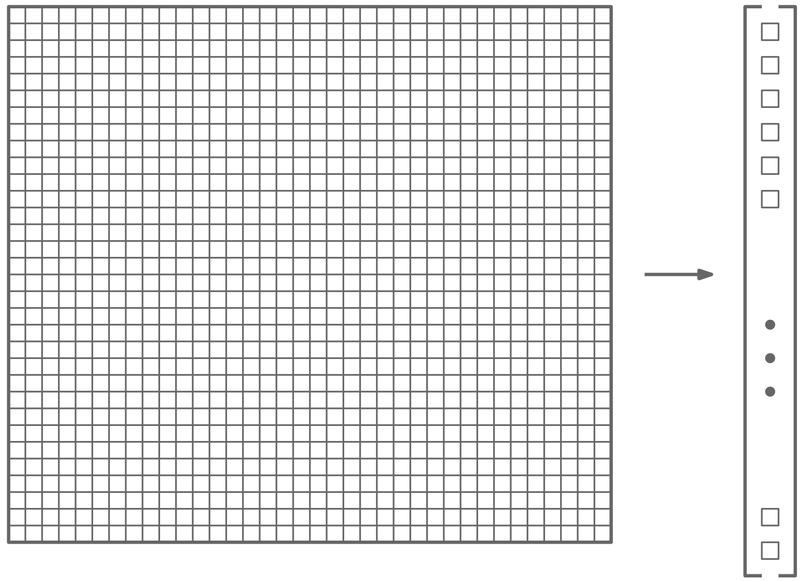
So far…

- We played around with images…



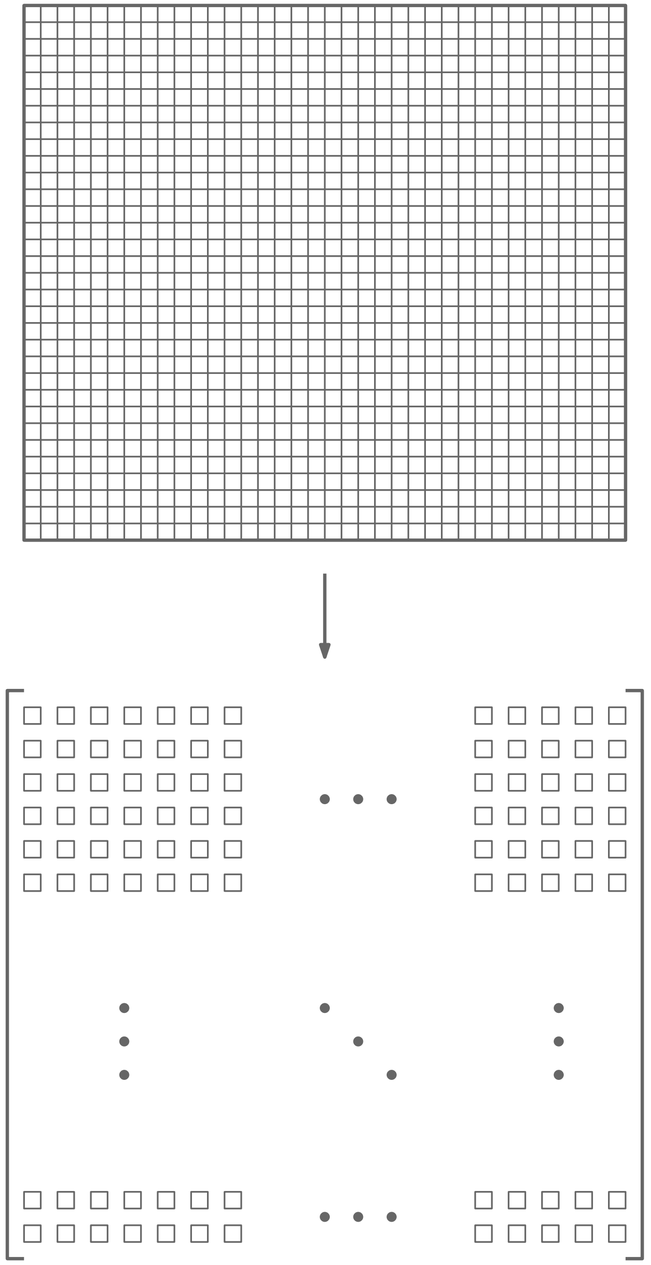
So far…

- We played around with images…



So far…

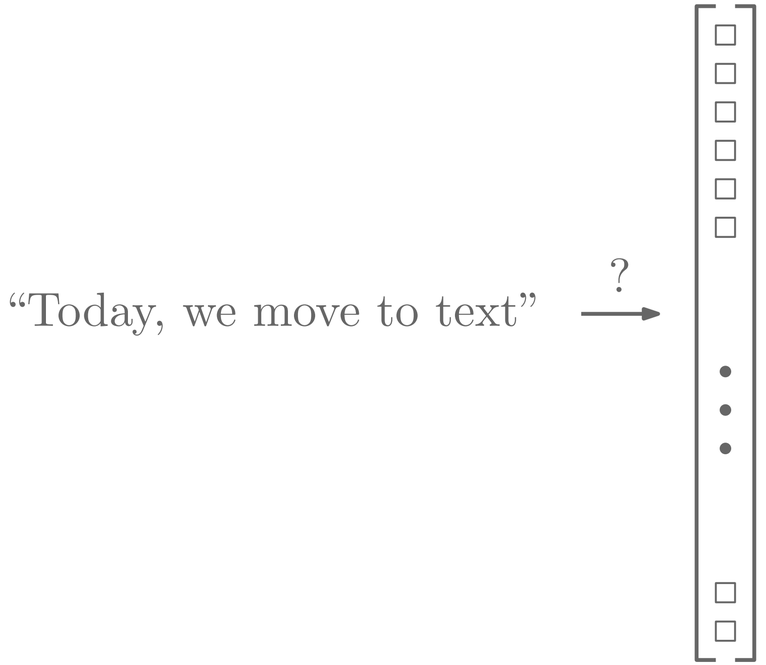
- We played around with images…



Today, we move to text

Problem n. 1

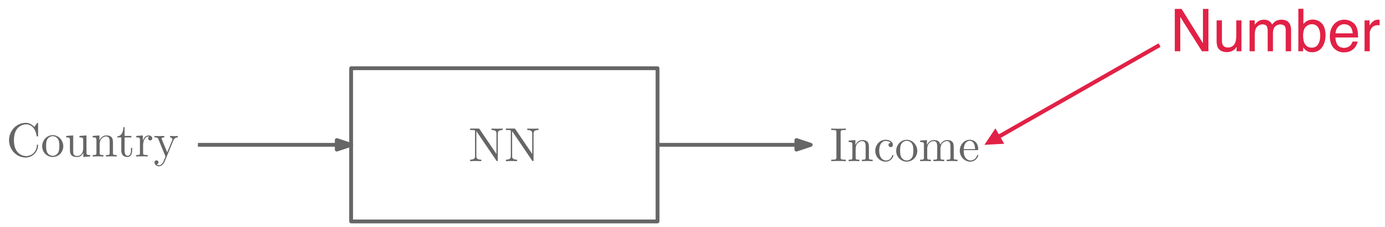
- How do we represent text as a vector/matrix of numbers?



Representations

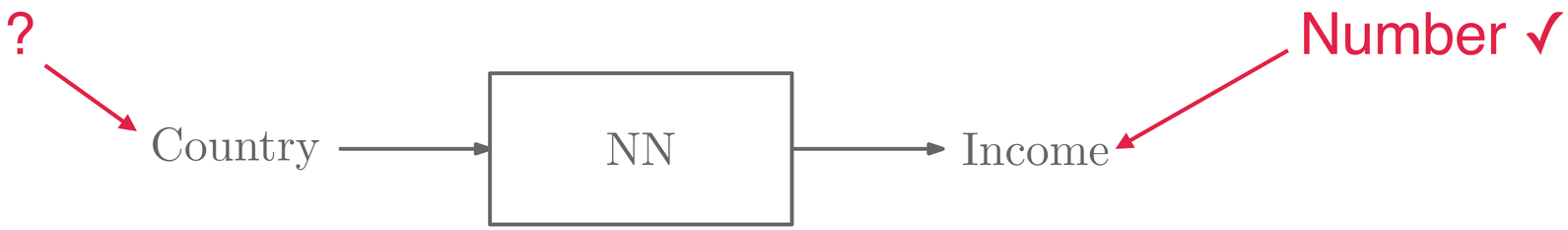
Representations

- Neural networks expect vectors/matrices of numbers as inputs
- How can we represent a non-numerical quantity as a vector/matrix of numbers?
- E.g.:
  - Suppose we want to predict average income given the country



Representations

- Neural networks expect vectors/matrices of numbers as inputs
- How can we represent a non-numerical quantity as a vector/matrix of numbers?
- E.g.:
  - Suppose we want to predict average income given the country



Idea n. 1

- Naive idea:
  - List all *N* countries
  - Assign to each country a number from 1 to *N*

1. Afghanistan
2. Albania
3. Algeria
4. Andorra
5. Angola
6. Antigua and Barbuda
7. Argentina

… …

Problem:

The numbers have no numerical meaning

Idea n. 2

- One-hot encoding
  - List all *N* countries
  - Represent each country as a *N*-element vector with a single 1

Afghanistan

Albania

Algeria

Andorra

Angola

Antigua and Barbuda

Argentina

…

Idea n. 2

One-hot encoding:

We can interpret these numbers as “probabilities”

1	0	0	0	0	0	0	…	0
0	1	0	0	0	0	0	…	0
0	0	1	0	0	0	0	…	0
0	0	0	1	0	0	0	…	0
0	0	0	0	1	0	0	…	0
0	0	0	0	0	1	0	…	0
0	0	0	0	0	0	1	…	0
…	…	…	…	…	…	…	…	…

Afghanistan

Albania

Algeria

Andorra

Angola

Antigua and Barbuda

Argentina

…

What about text?

- List all possible words in our vocabulary
- Represent each word using one-hot encoding

One-hot encoding

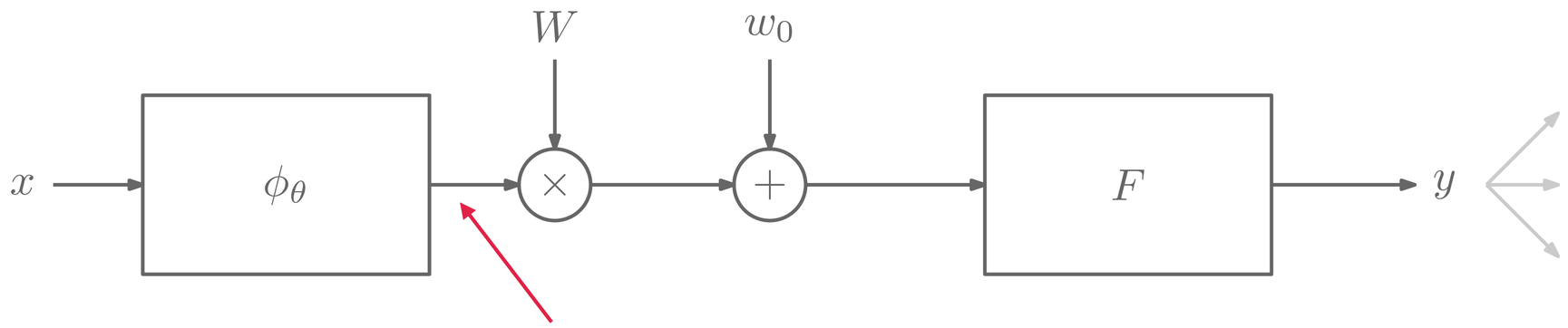
- Works fine for text
- However…
  - One-hot encoding may lead to very large representations

    (1,000,000 words → 1,000,000 element vector)
  - Representations are local (one dimension per word)

Can we do better?

Learning representations

- Neural networks are good at learning representations

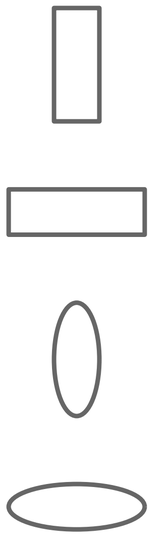


Learning representations

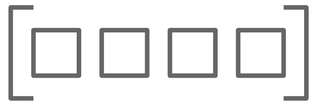
- Neural networks are good at learning representations
- We can try to learn a better (non-local) representation for our input

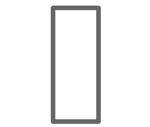
Local vs distributed representations

No pattern

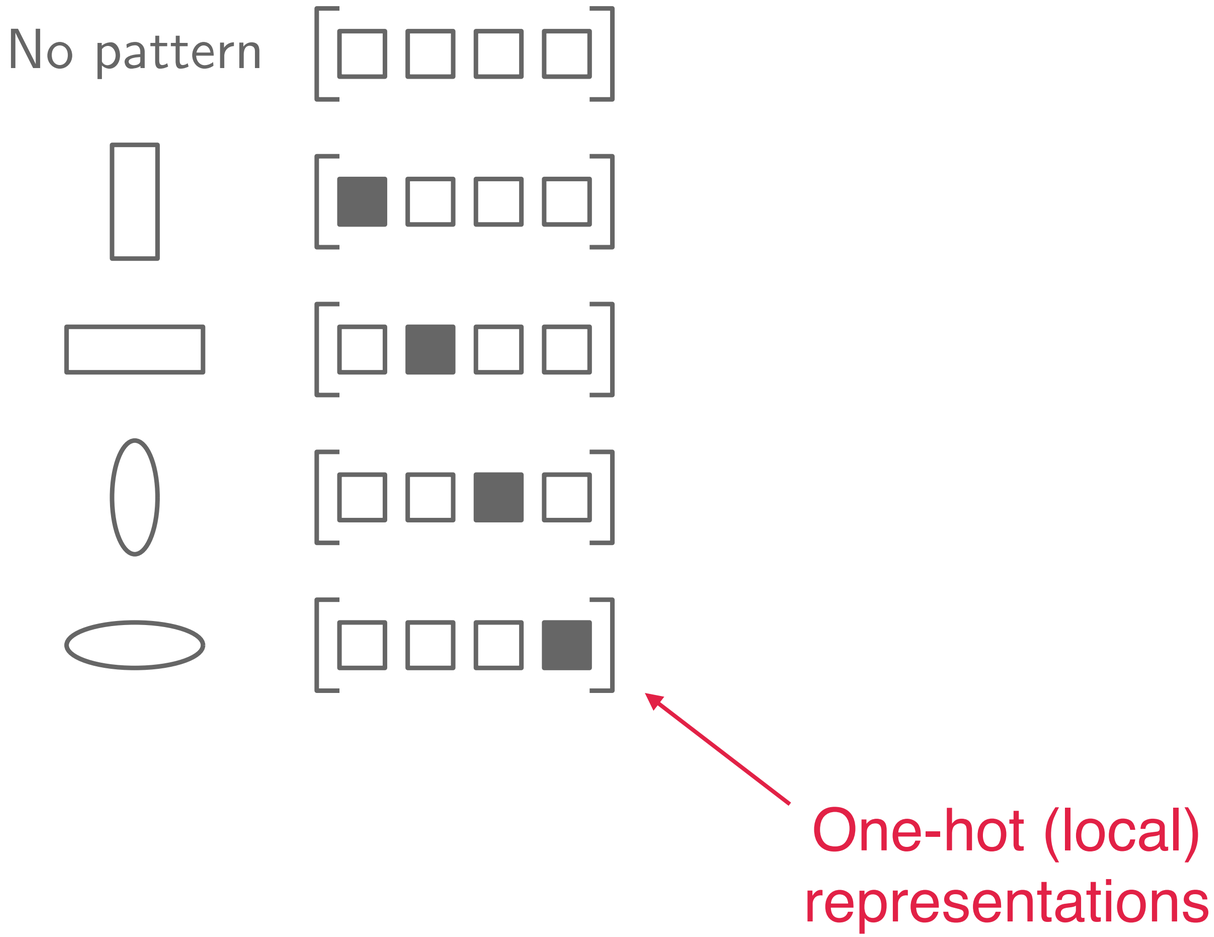


Local vs distributed representations

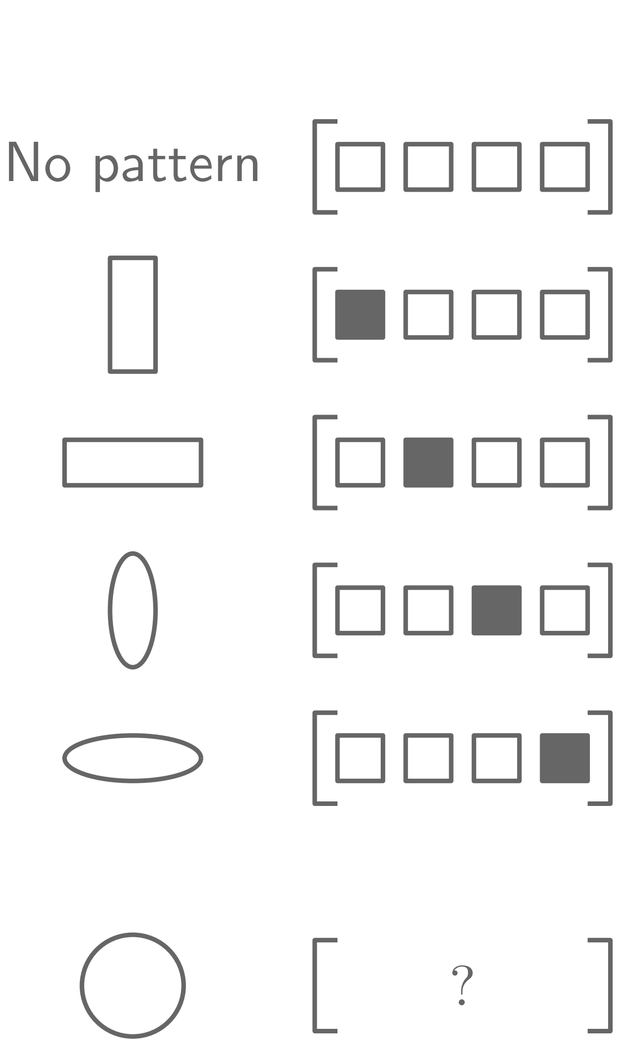
No pattern 

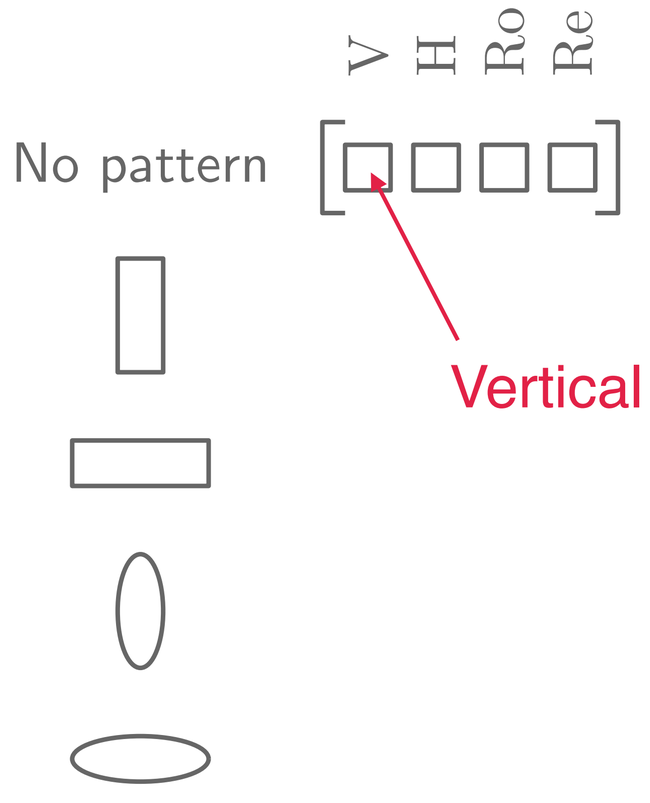


Local vs distributed representations

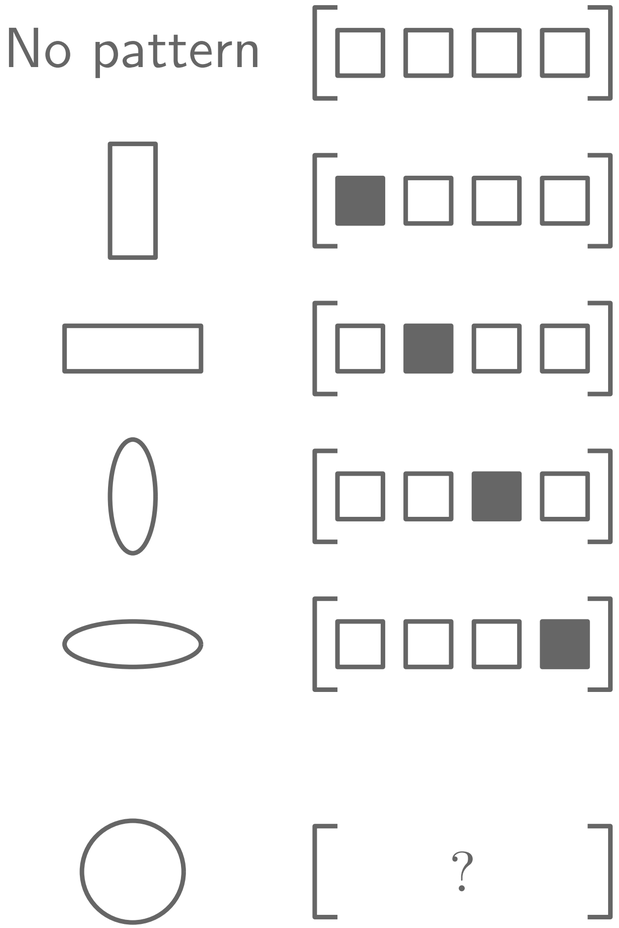


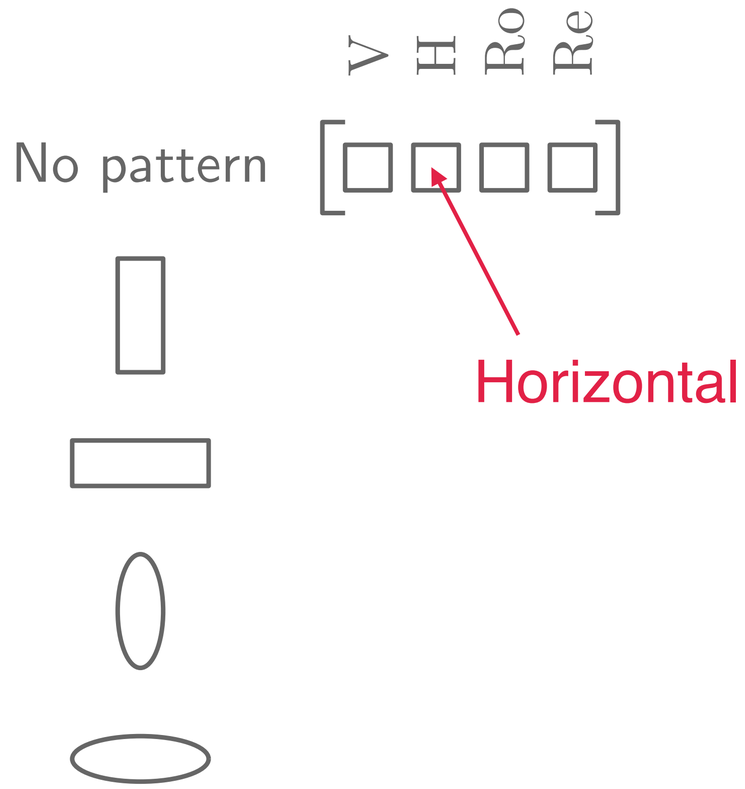
Local vs distributed representations



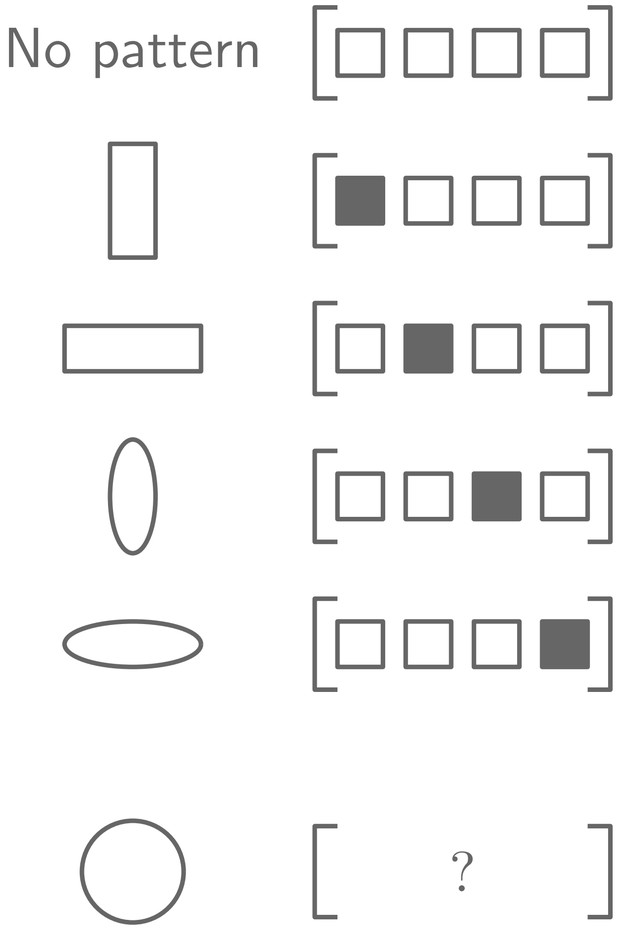


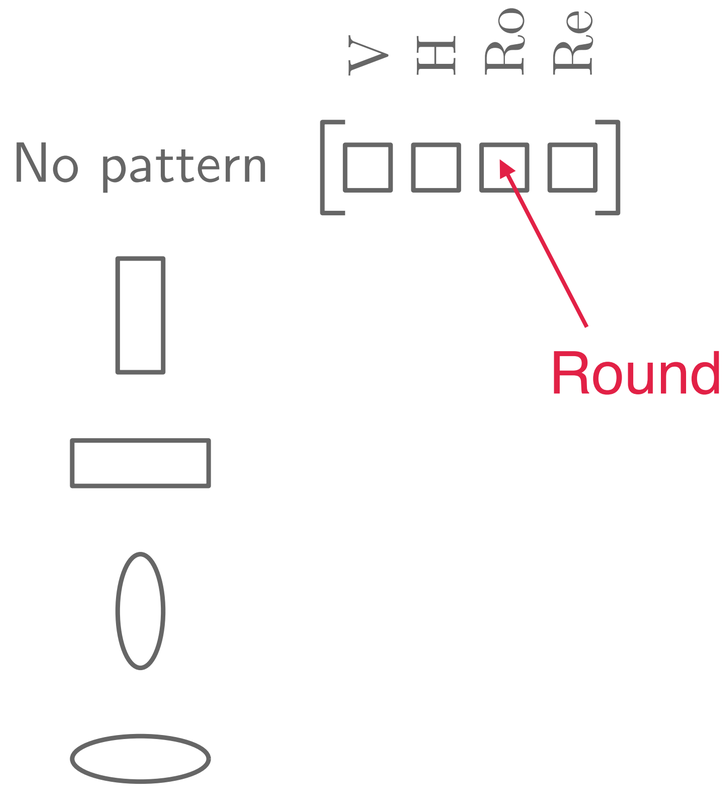
Local vs distributed representations



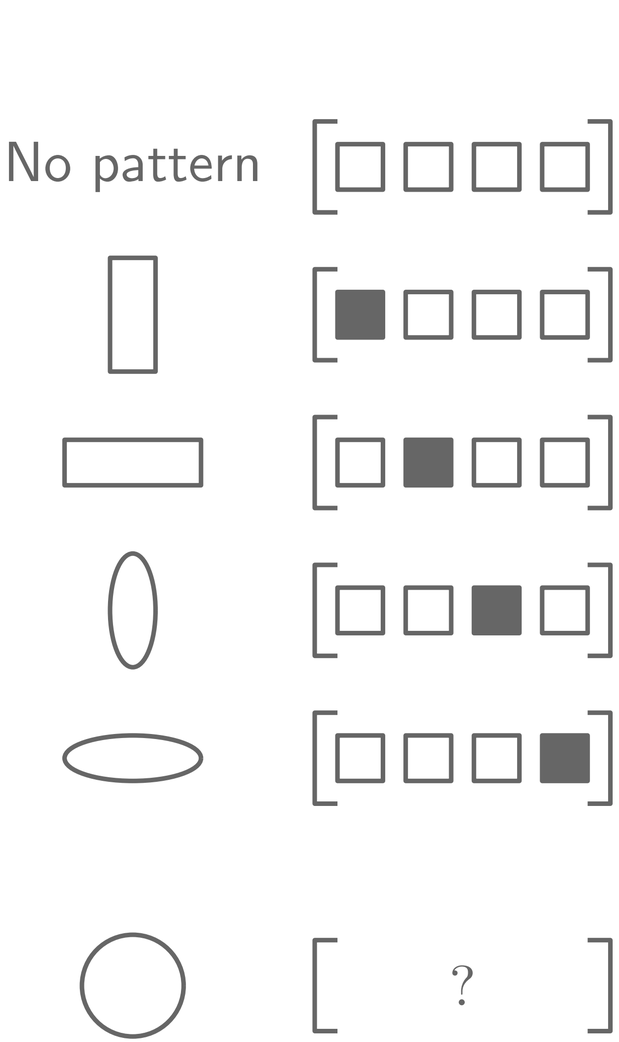


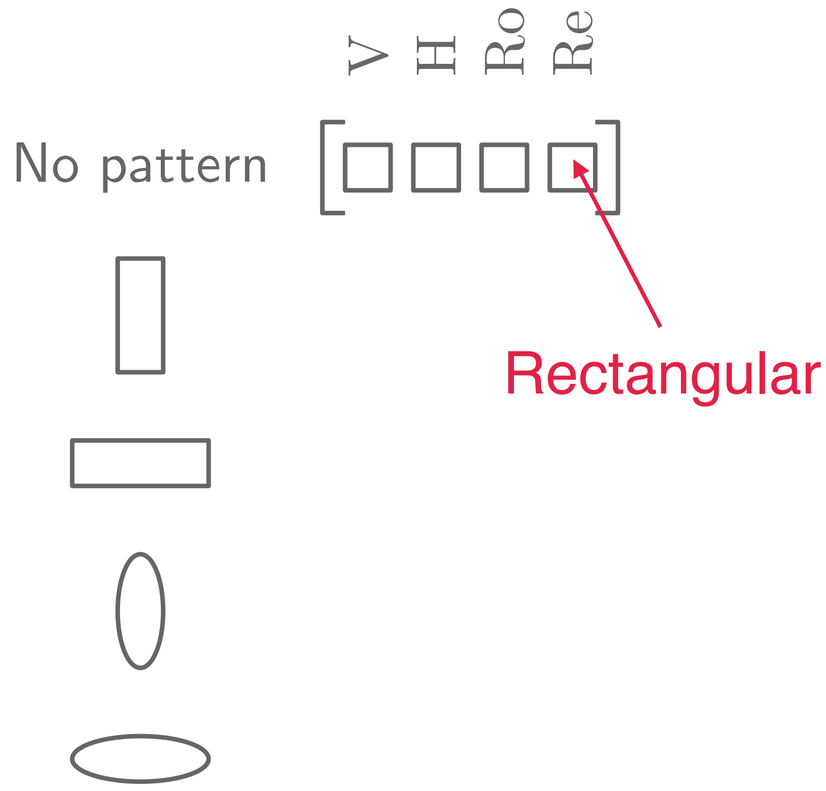
Local vs distributed representations



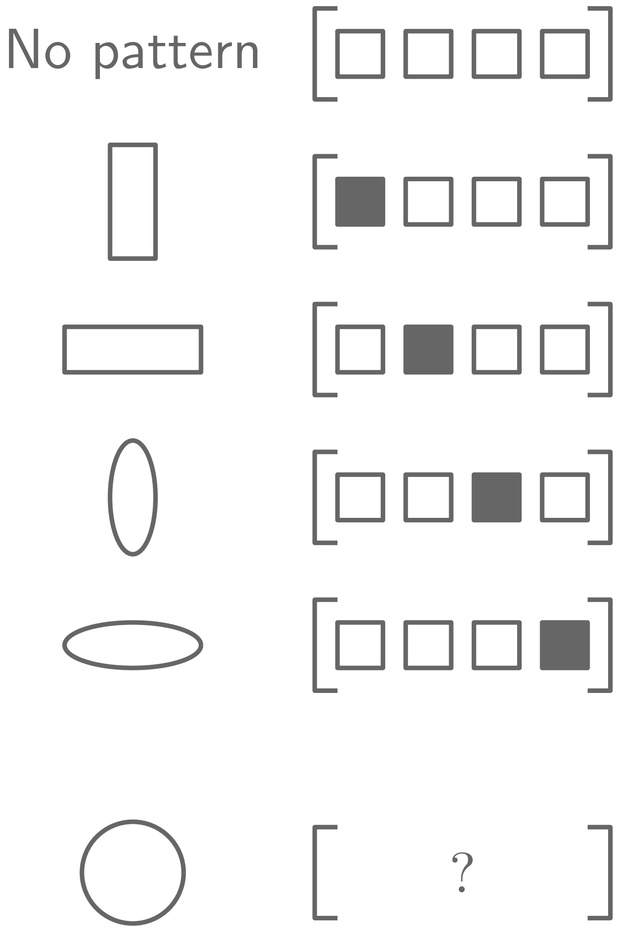


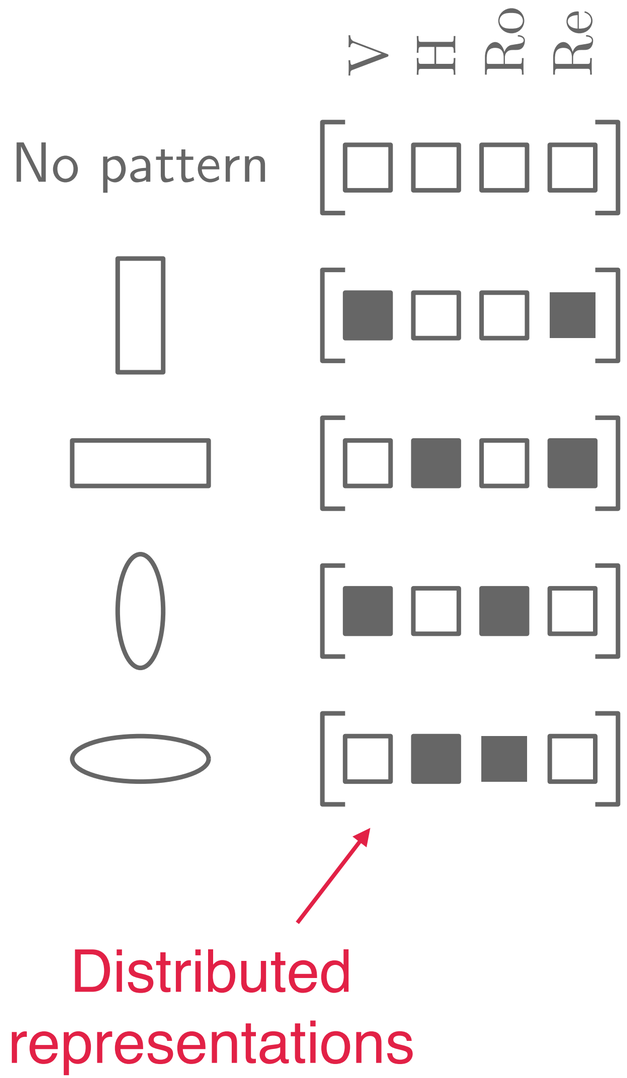
Local vs distributed representations



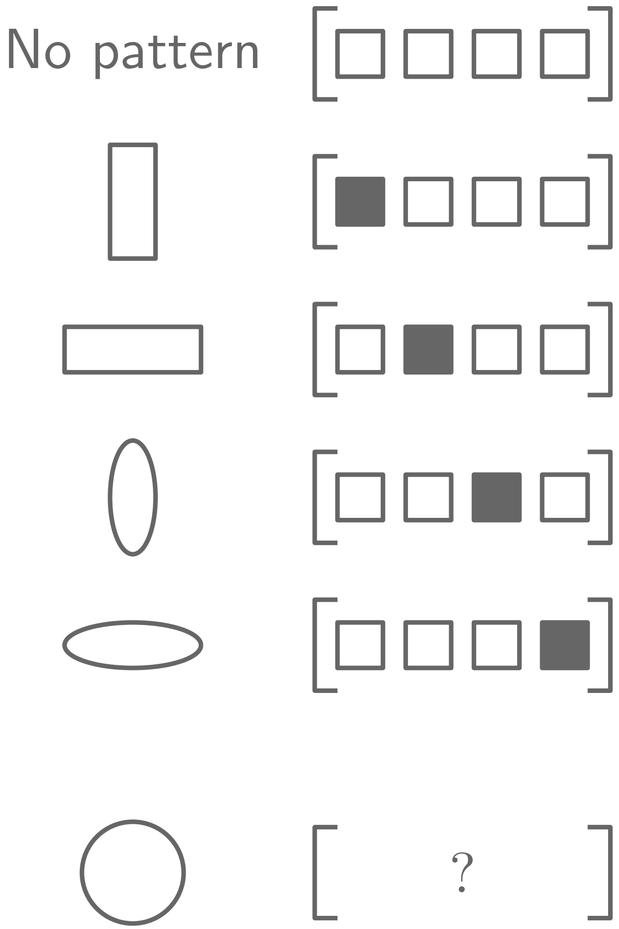


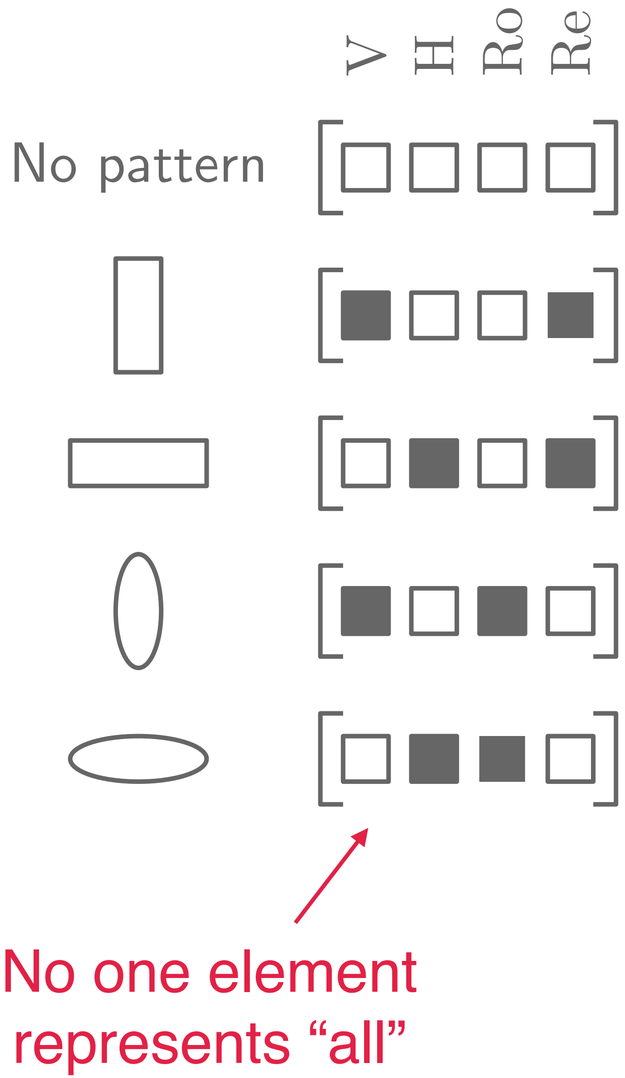
Local vs distributed representations



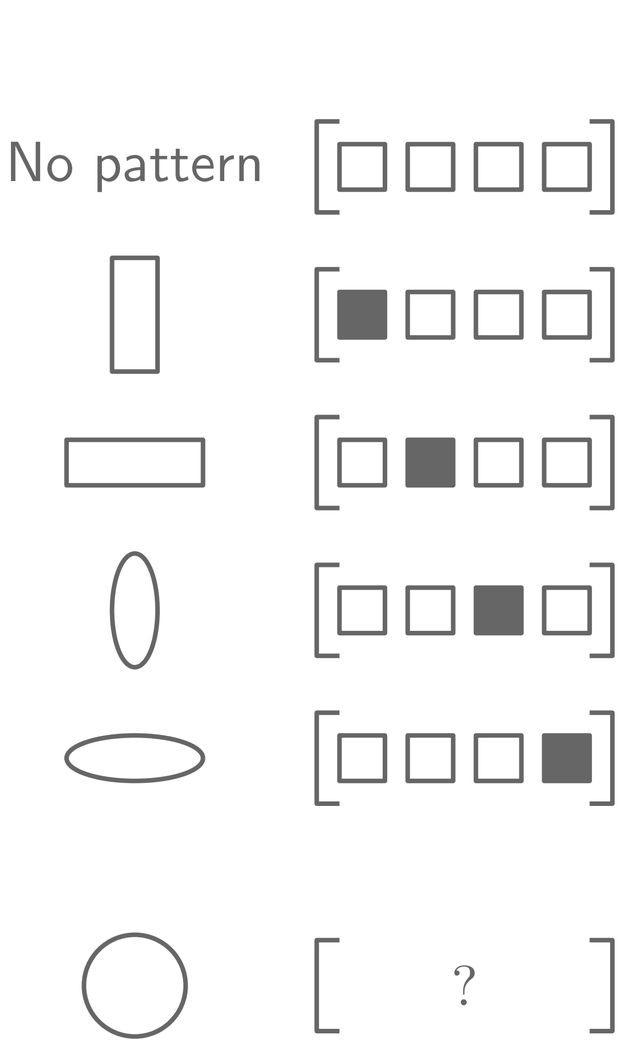


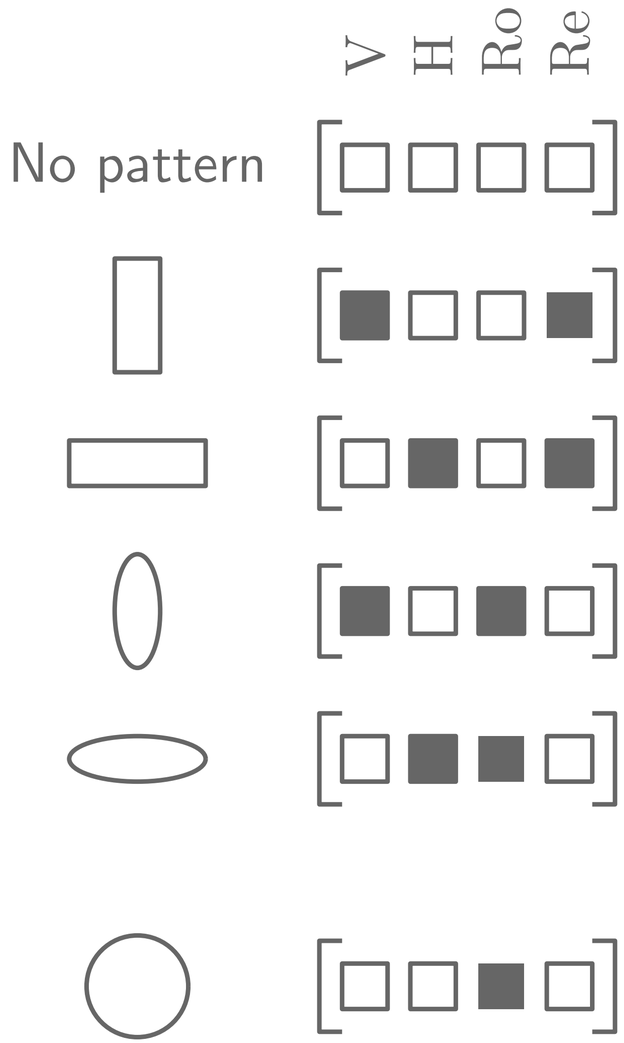
Local vs distributed representations





Local vs distributed representations





The power of distributed representations

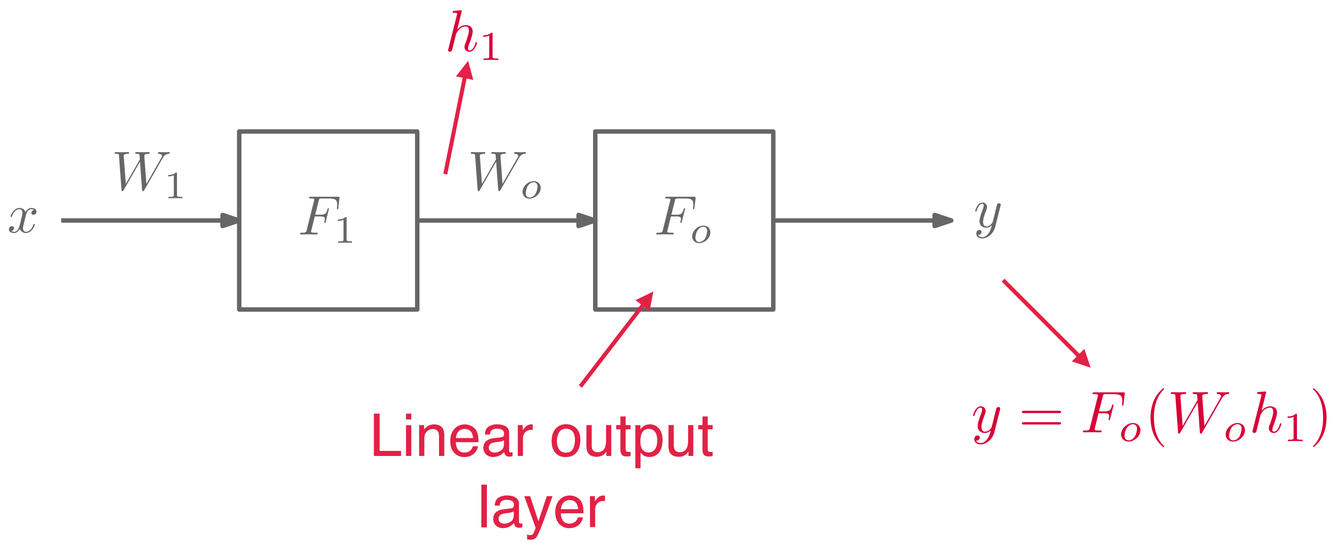
- Distributed representations are:
  - More compact (a vector with N attributes may be used to represent up to $O(2^N)$ objects)
  - More powerful (may generalize in a meaningful way)

Learning representations

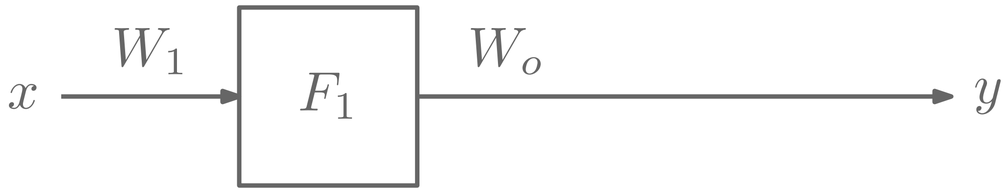
- What is a good representation?
  - Retains all the relevant information
  - Compact

- Can we have a neural network learn to do that?

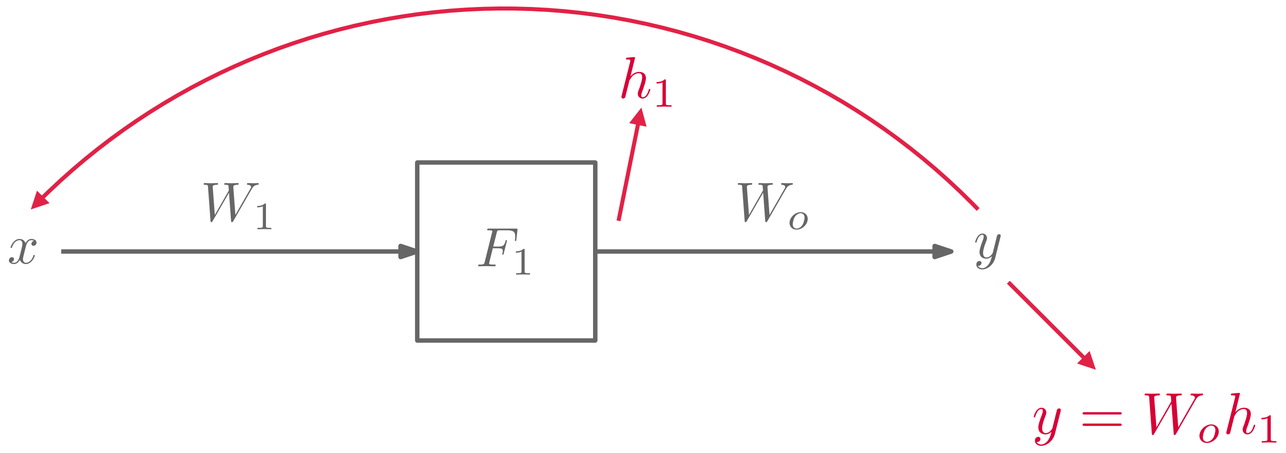
Learning representations



Learning representations

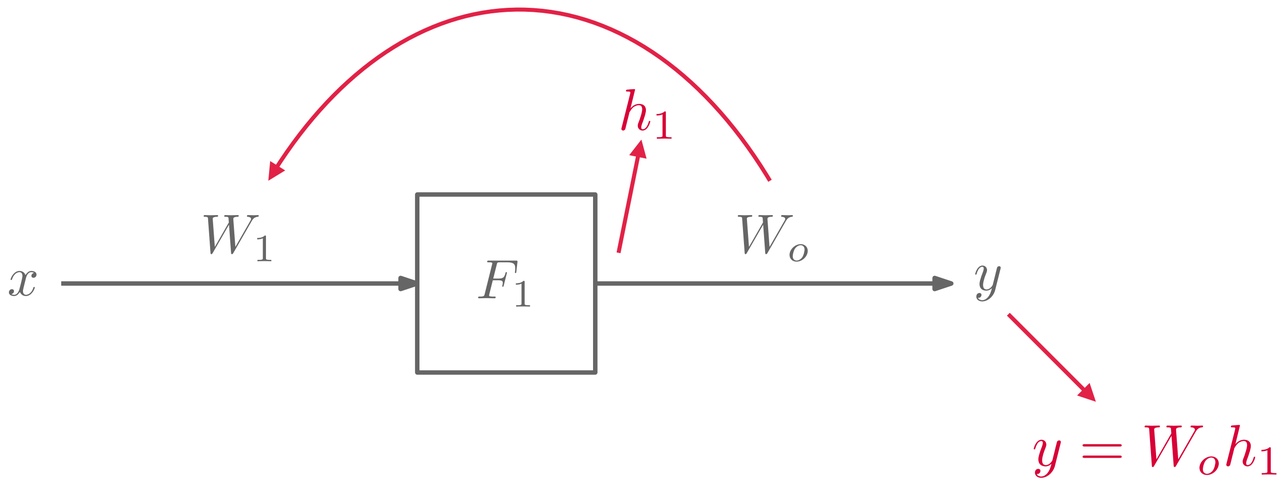


Learning representations



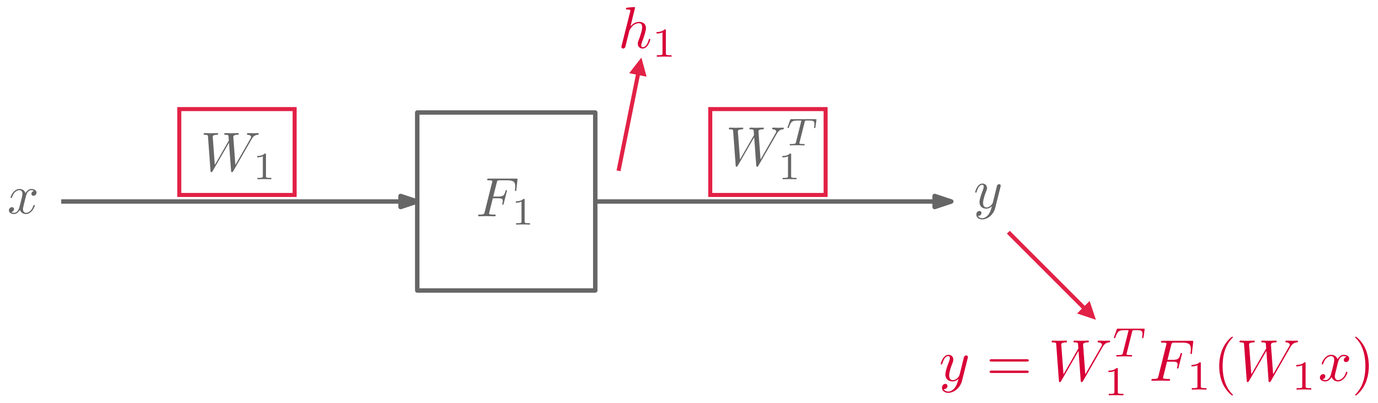
1. Capturing all relevant information → We should be able to reconstruct the input *x* → Goal: Minimize distance between *y* and *x*, $\|y - x\|^2$

Learning representations



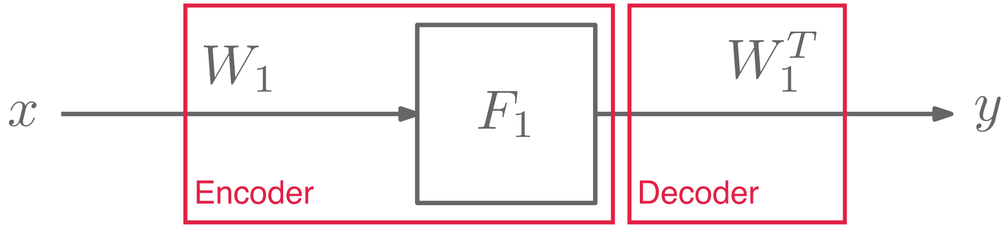
1. Capturing all relevant information → We should be able to reconstruct the input *x* → Force W_o and W_1 to be similar

Learning representations



2. Compact ⟶ We make dim(h_1) < dim(x) ⟶ W_1 is $m \times n$ with $m < n$

Auto-encoder



- We train the neural network so that it reconstructs the input in the output
- Encoder computes a compact representation:

$$h_1 = F_1(W_1 x)$$

Code

- Decoder reconstructs input from learned representation:

$$y = W_1^T h_1$$

Interesting facts

- If F_1 is linear and we use the squared loss, the optimal auto-encoder is the same as PCA
- Auto-encoders were historically used for unsupervised pre-training of deep NNs:
  - (Pre-)Train one layer at a time as an auto-encoder
  - Once all hidden layers are pre-trained, add output layer and train everything as usual

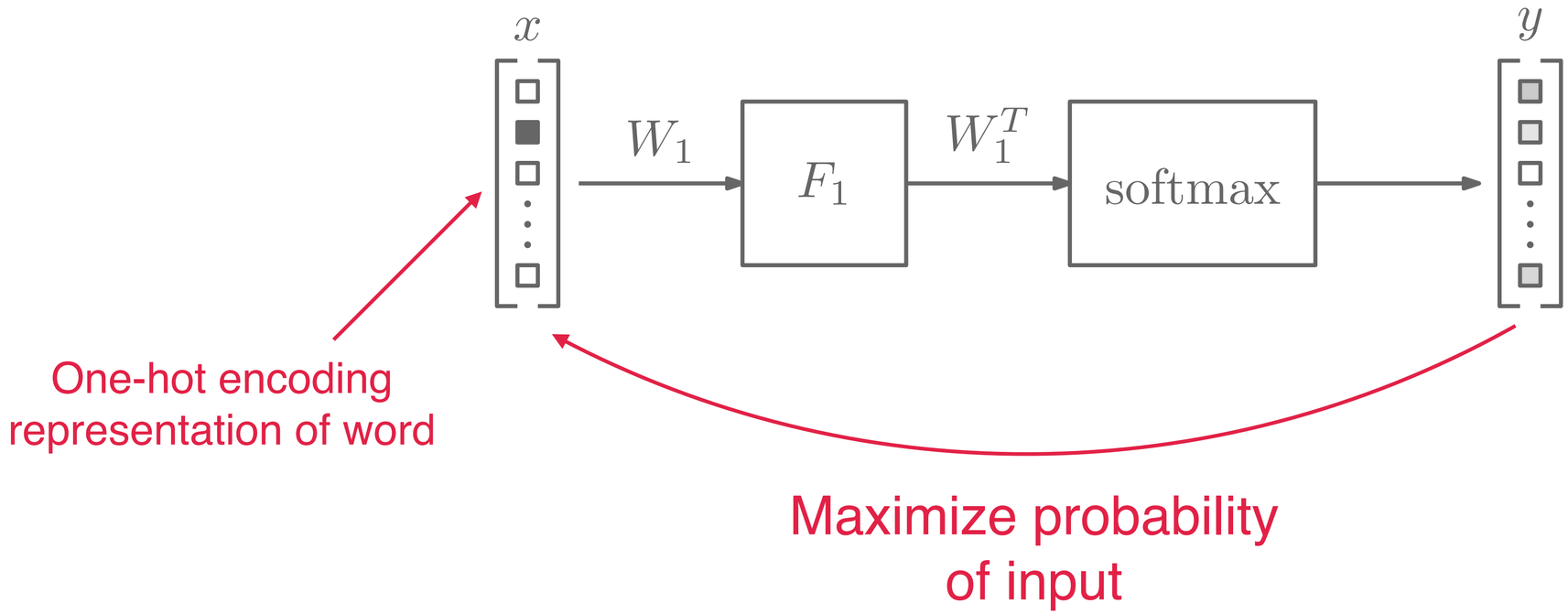
Variations of auto-encoders

- Sparse auto-encoders: Add L^1 regularization to encourage sparse representations
- Denoising auto-encoders: Add noise to the input, training the AE to reconstruct denoised input
- Variational auto-encoders: Learn a probabilistic generative model (more on this in the last lecture)

Now, we move to text

Learning representations of text

- How can we apply these ideas to learn good representations of text/words?



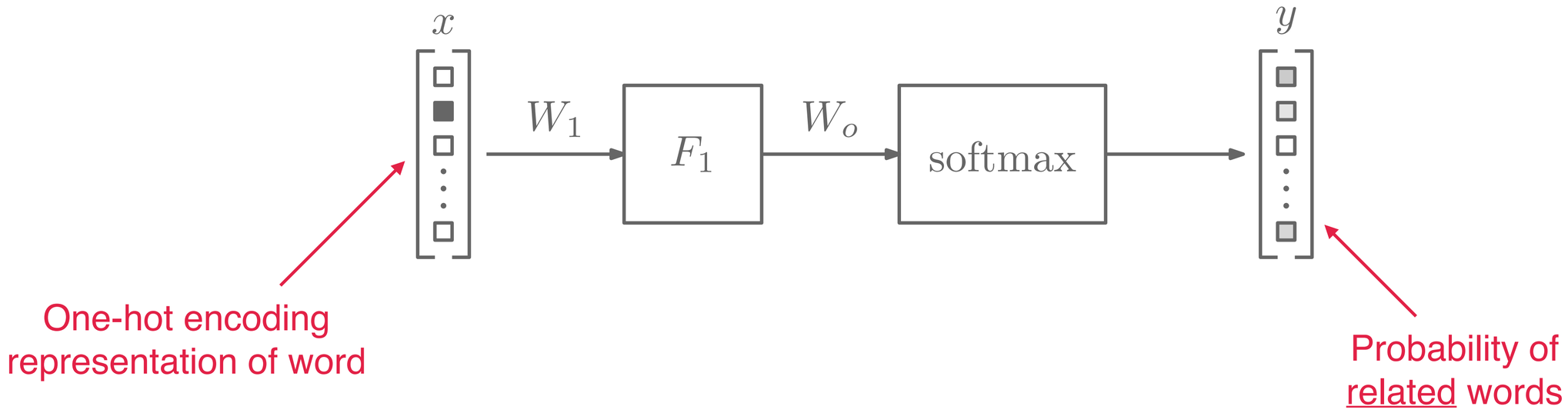
Learning representations of text

- Such naive approach learns “arbitrary” representations for words
- Is it possible that semantically related words have related representations?

Context

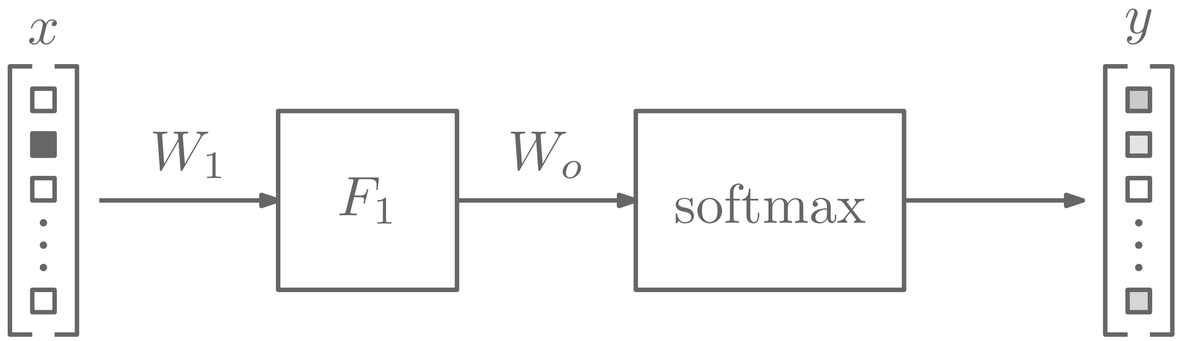
Learning representations of text

- Idea: Represent a word in terms of its ability to predict its context



Learning representations of text

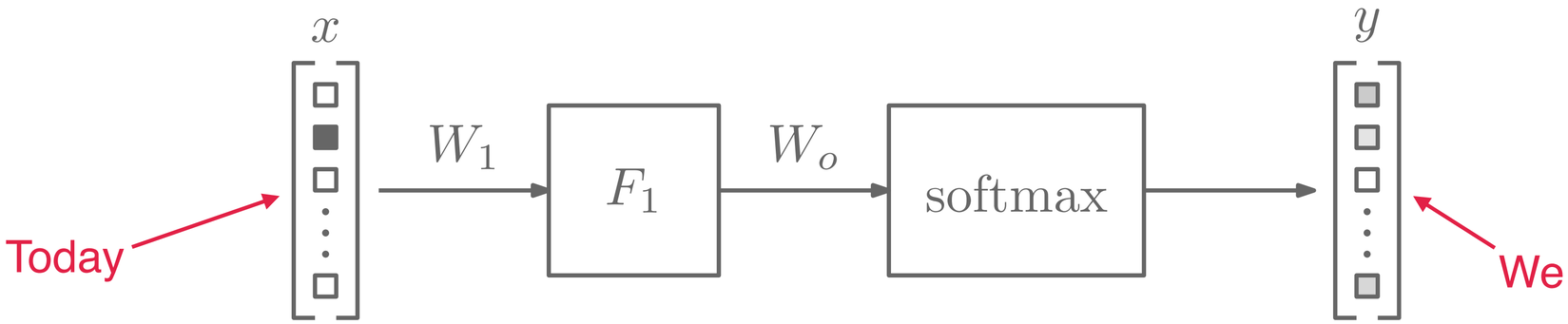
- Idea: Represent a word in terms of its ability to predict its context



Today we move to text

Learning representations of text

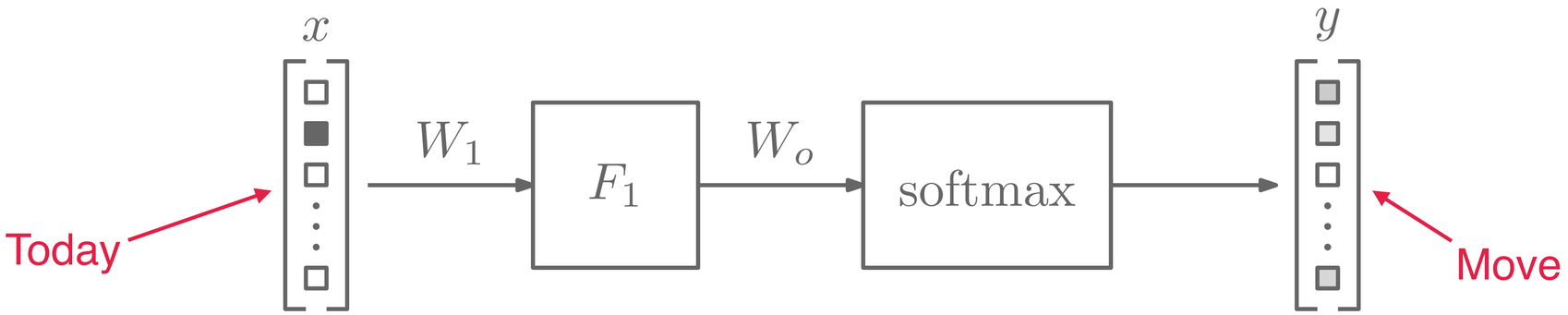
- Idea: Represent a word in terms of its ability to predict its context



Today	we	move	to	text

Learning representations of text

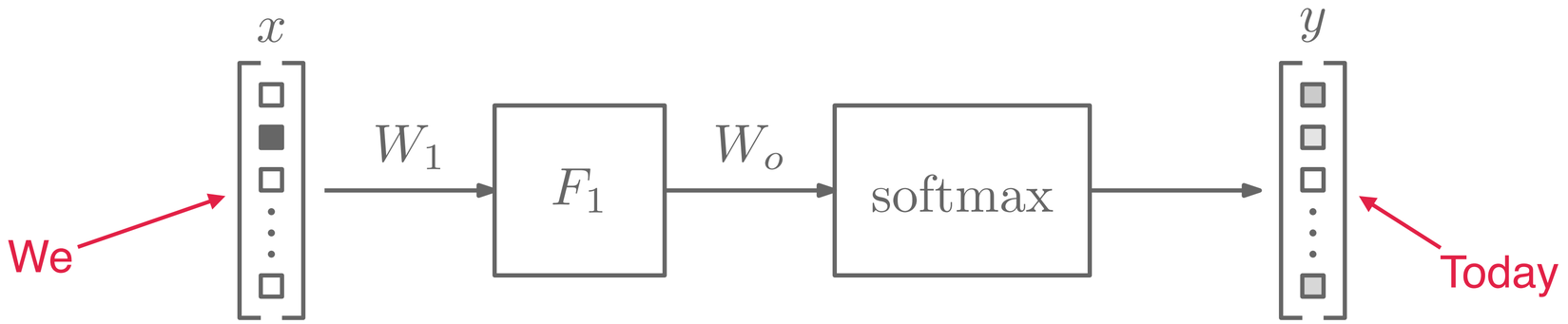
- Idea: Represent a word in terms of its ability to predict its context



Today	we	move	to	text

Learning representations of text

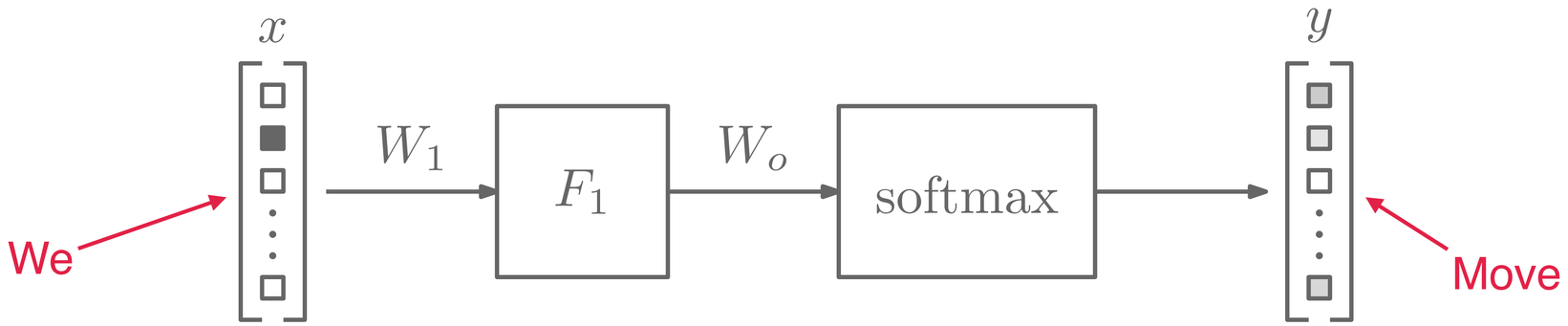
- Idea: Represent a word in terms of its ability to predict its context



Today	we	move	to	text

Learning representations of text

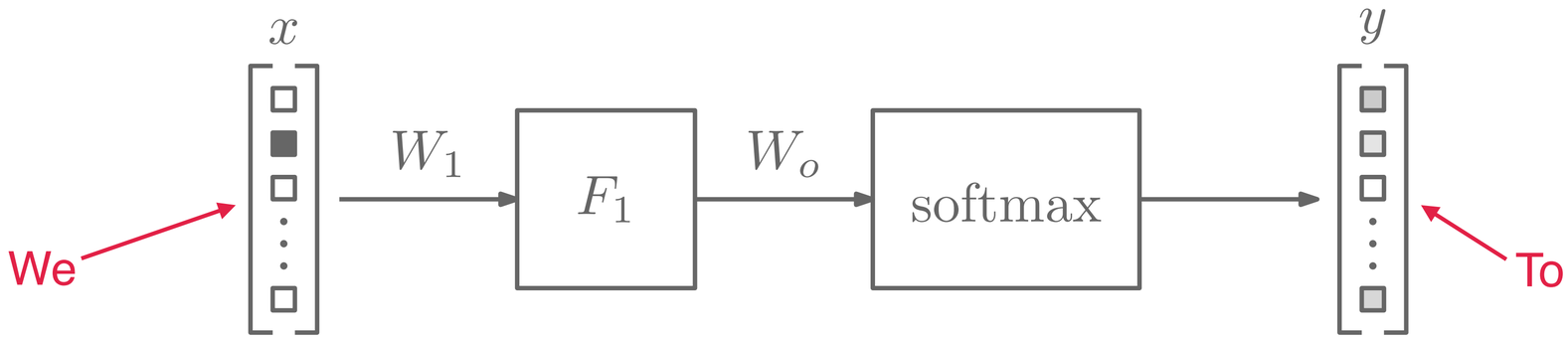
- Idea: Represent a word in terms of its ability to predict its context



Today	we	move	to	text

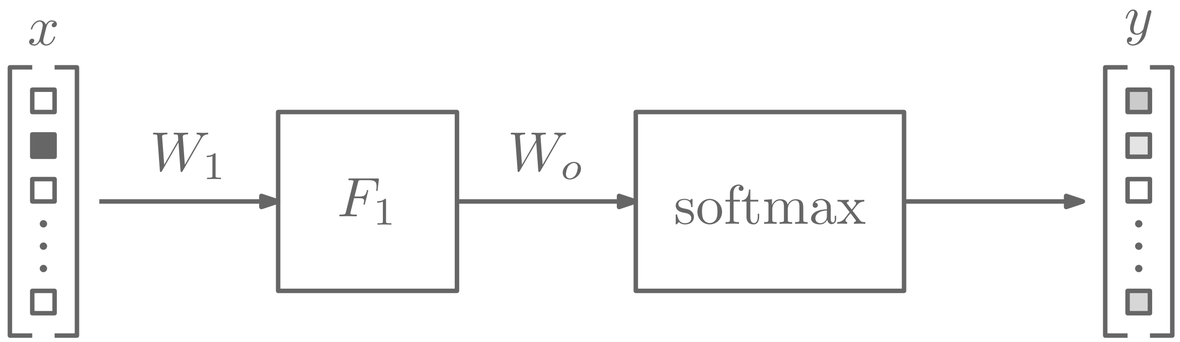
Learning representations of text

- Idea: Represent a word in terms of its ability to predict its context



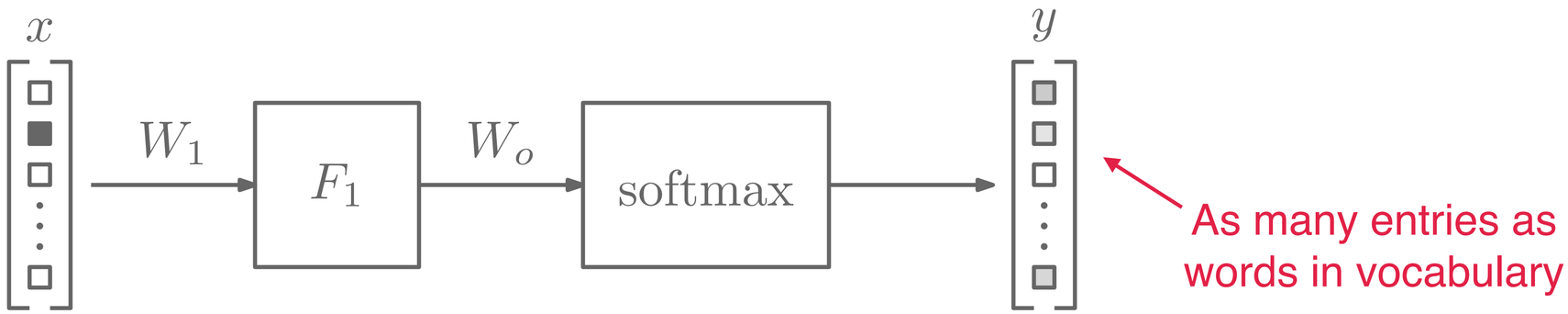
Today	we	move	to	text

Word2vec



- The previous approach is known as word2vec (skipgram model)
- Typically used with linear F_1
- The resulting representation is known as a word embedding
  - The word embedding of the ith word in the vocabulary corresponds to the ith column of W_1

Negative sampling in word2vec



- A difficulty in the implementation of the word2vec model above is the size of the softmax for large vocabularies
- If model is trained with softmax output, training will be very slow

Negative sampling in word2vec

- Output of the word2vec network:

$$[y]_c = \frac{\exp(s_\theta(c, w))}{\sum_{c'} \exp(s_\theta(c', w))} \approx p(c \mid w)$$

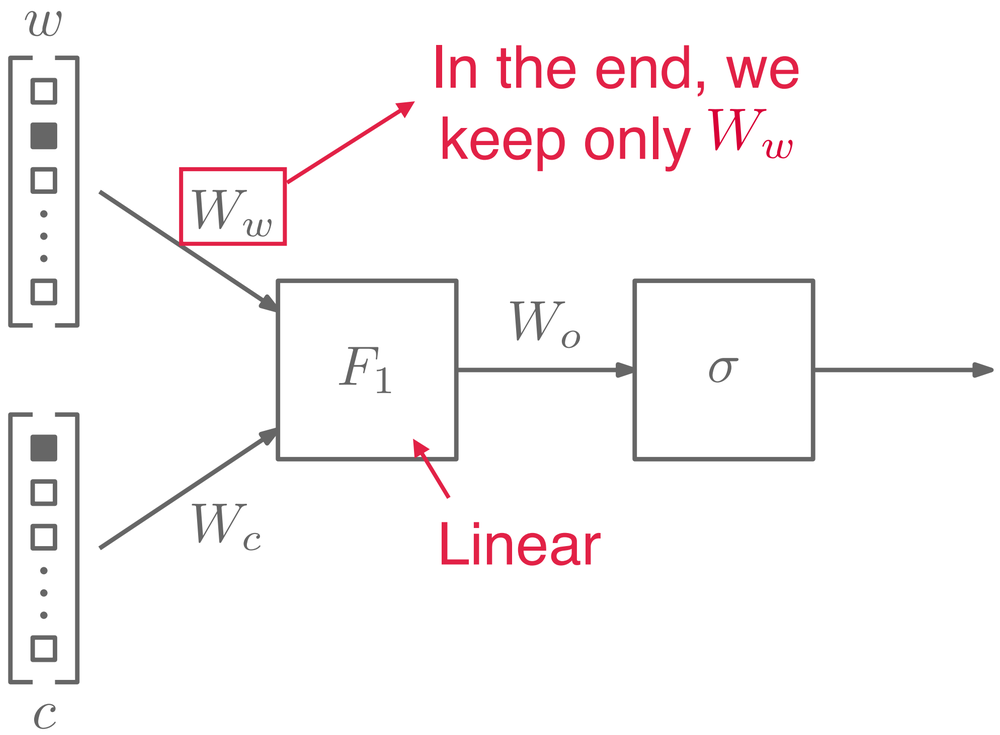
Summation over all vocabulary

Our target distribution

Negative sampling in word2vec

- Suppose that:
  - We sample a pair (*c*, *w*) from the dataset
  - We build pairs (*c*', *w*), where *c*' is selected randomly from the vocabulary
- We train a binary classifier to distinguish “true pairs” from “false pairs”

Negative sampling in word2vec



In the end, we keep only W_w

Linear

$$y \approx \mathbb{P}[\text{pair is true} \mid (c, w)]$$

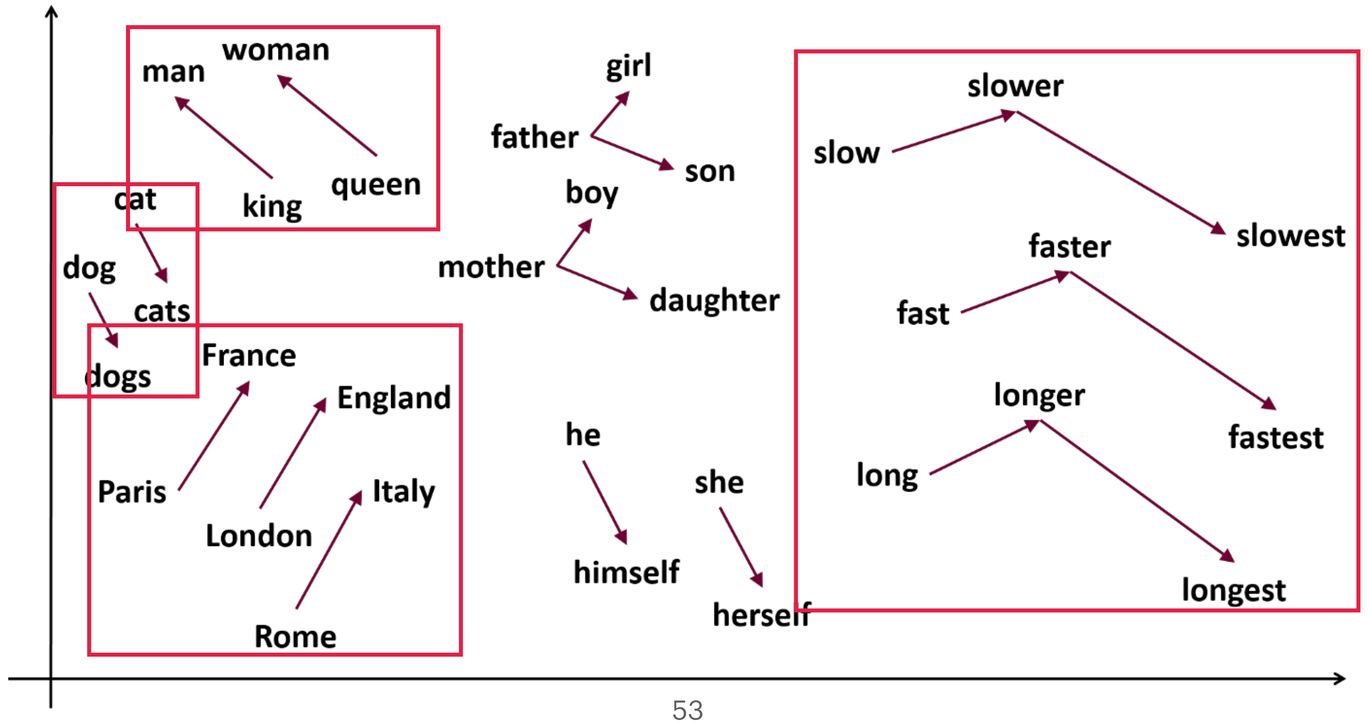
Approximately

$$\frac{1}{1 + \exp(-s_\theta(w, c))}$$

Same quantity as before

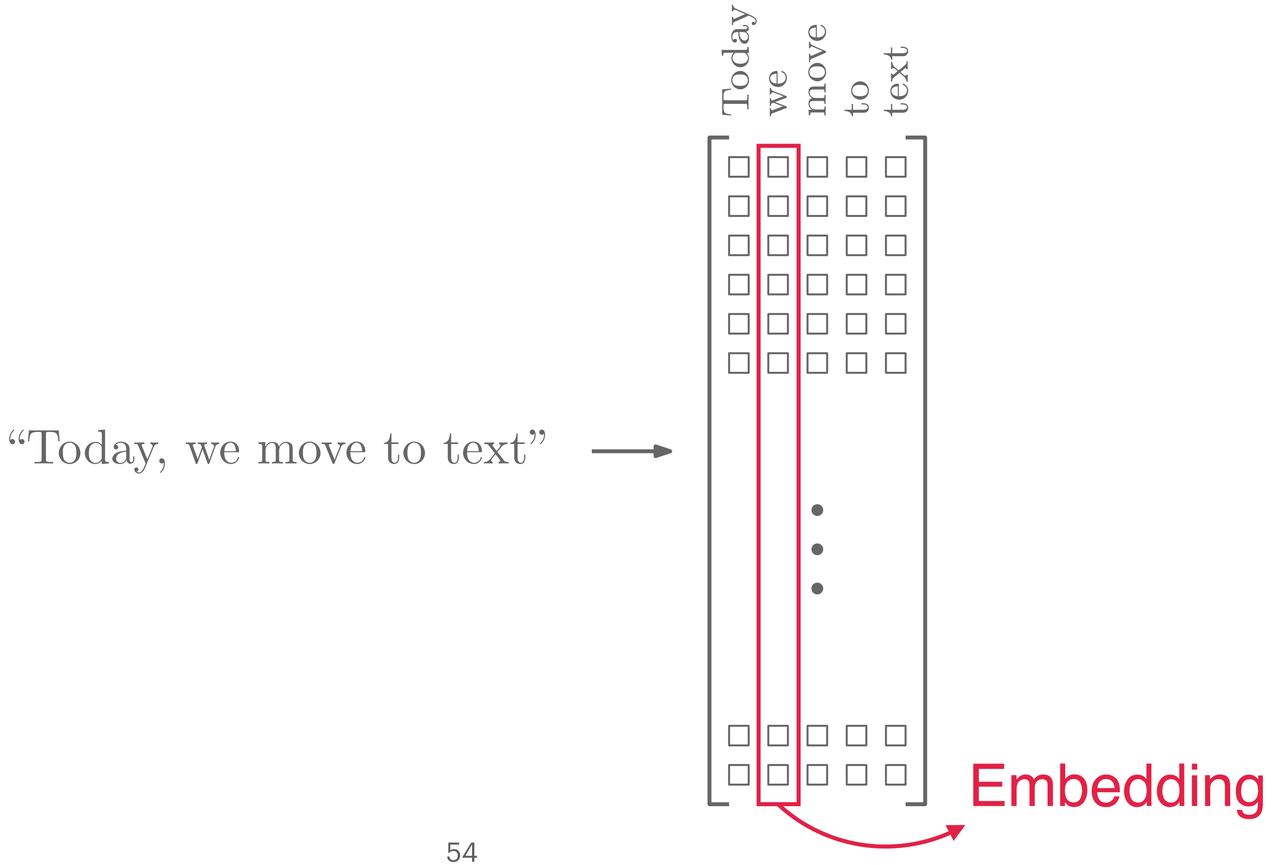
Word2vec

- We can visualize the representations learned by word2vec by projecting them into a low-dimensional (2D) space:



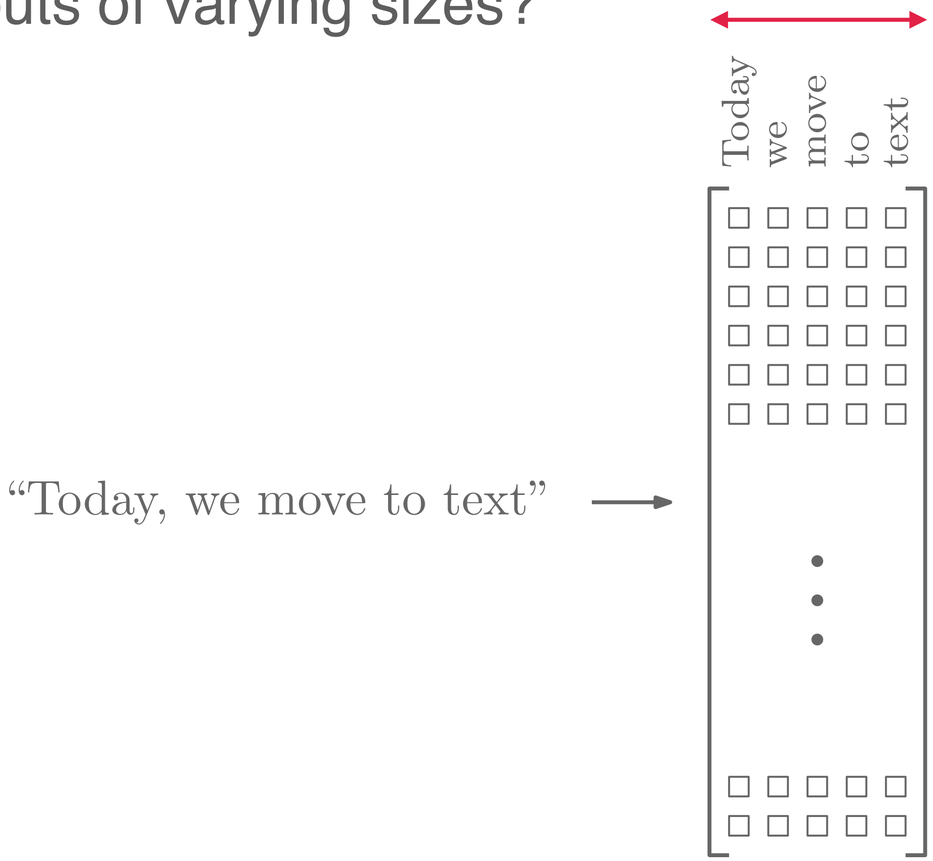
Problem n. 1

- How do we represent text as a vector/matrix of numbers?



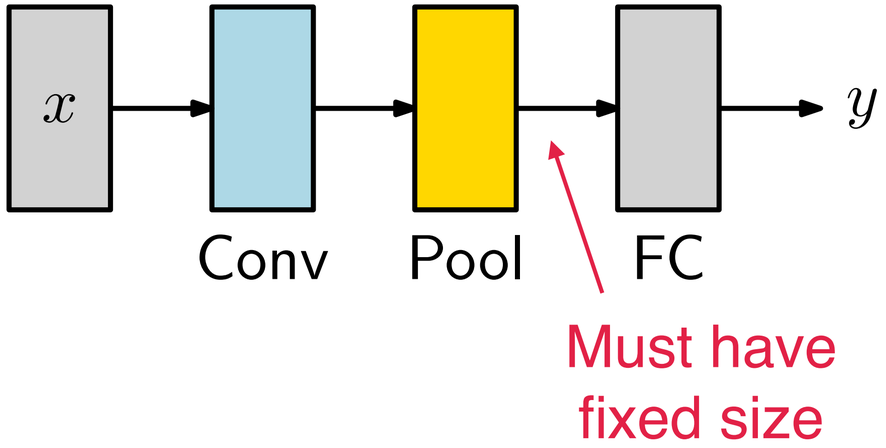
Problem n. 2

- How do we deal with inputs of varying sizes?

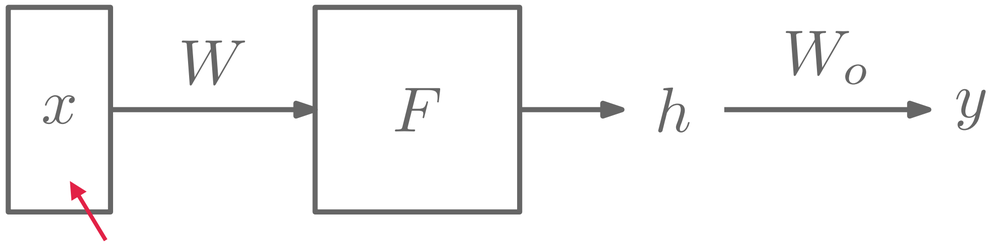


Recurrent neural networks

Convolutional neural networks

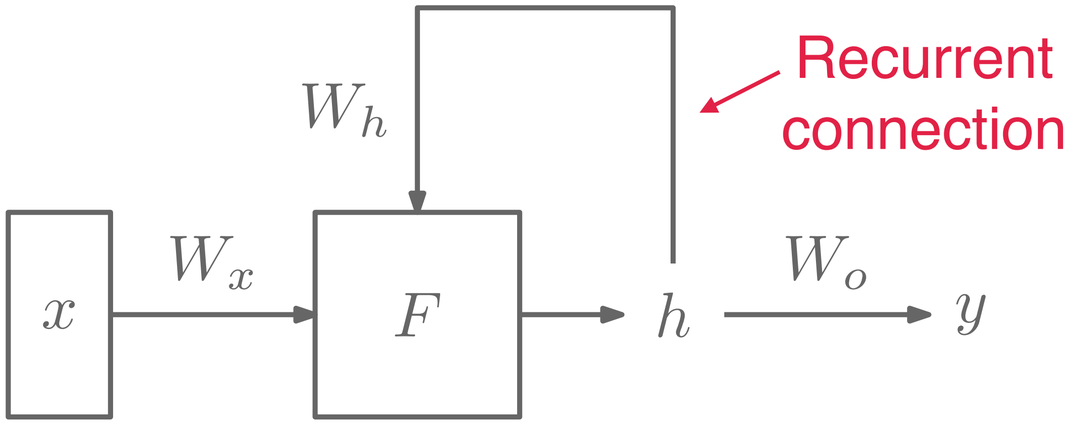


Feed-forward neural networks



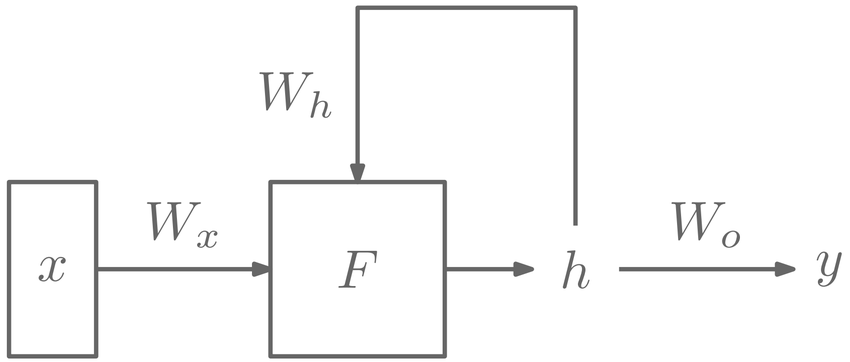
$$h = F(Wx)$$
$$y = W_o h$$

Recurrent neural networks



$$h = F(W_x x + W_h h)$$
$$y = W_o h$$

Recurrent neural networks



- The hidden state *h* stores information from previous inputs
- This means that inputs of varying length (e.g., sentences) are treated as sequences
- The neural network processes one element of the sequence at a time

Unrolling time in RNNs

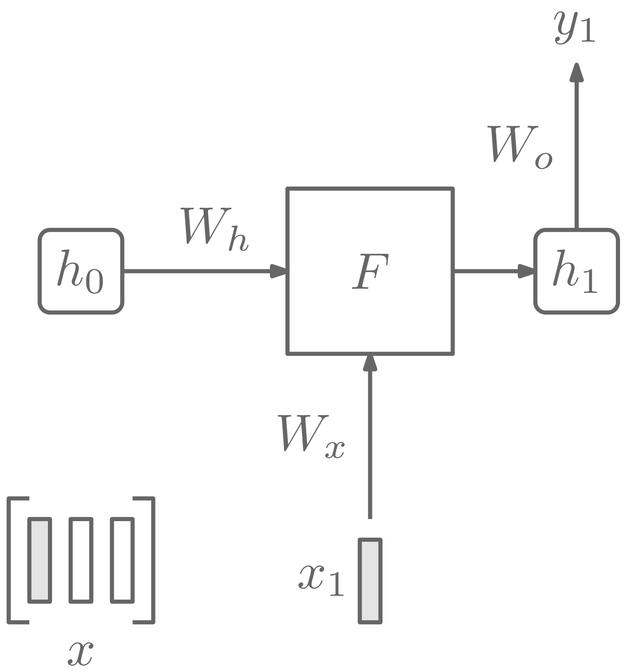
Initial hidden state

Input sequence

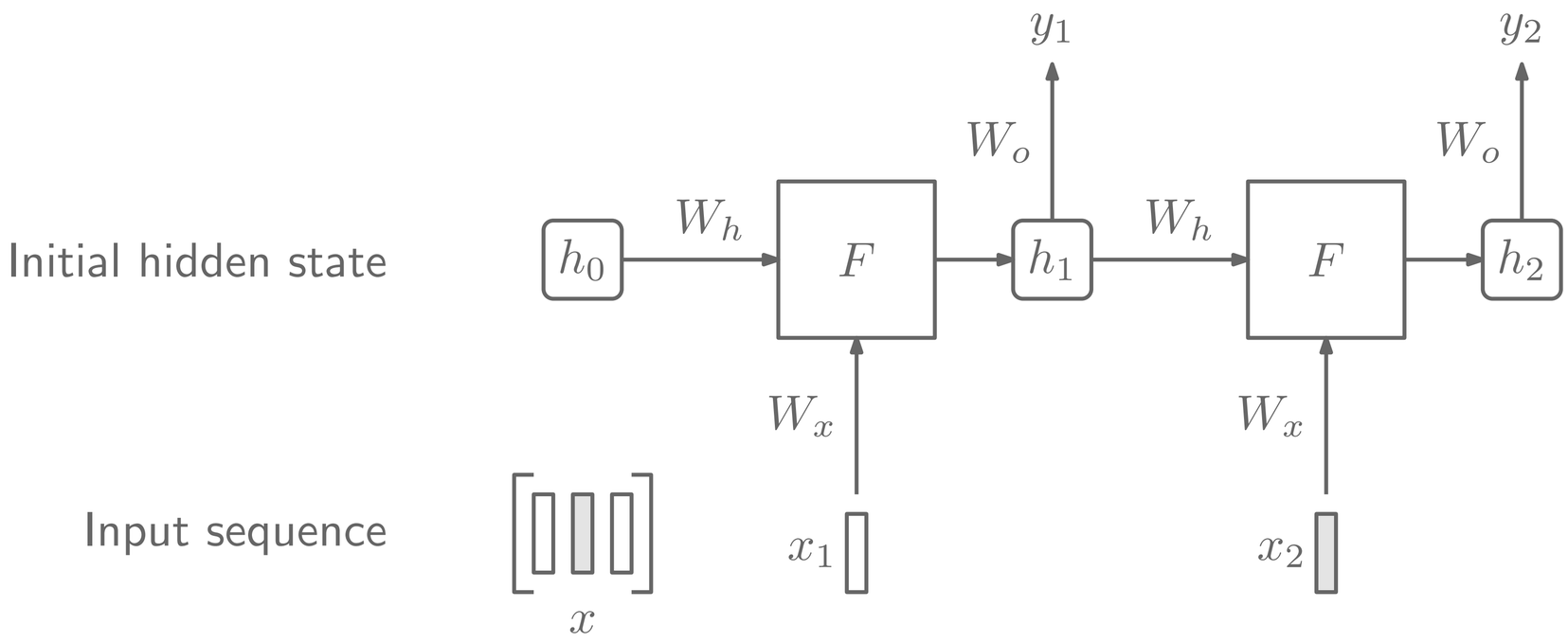
Unrolling time in RNNs

Initial hidden state

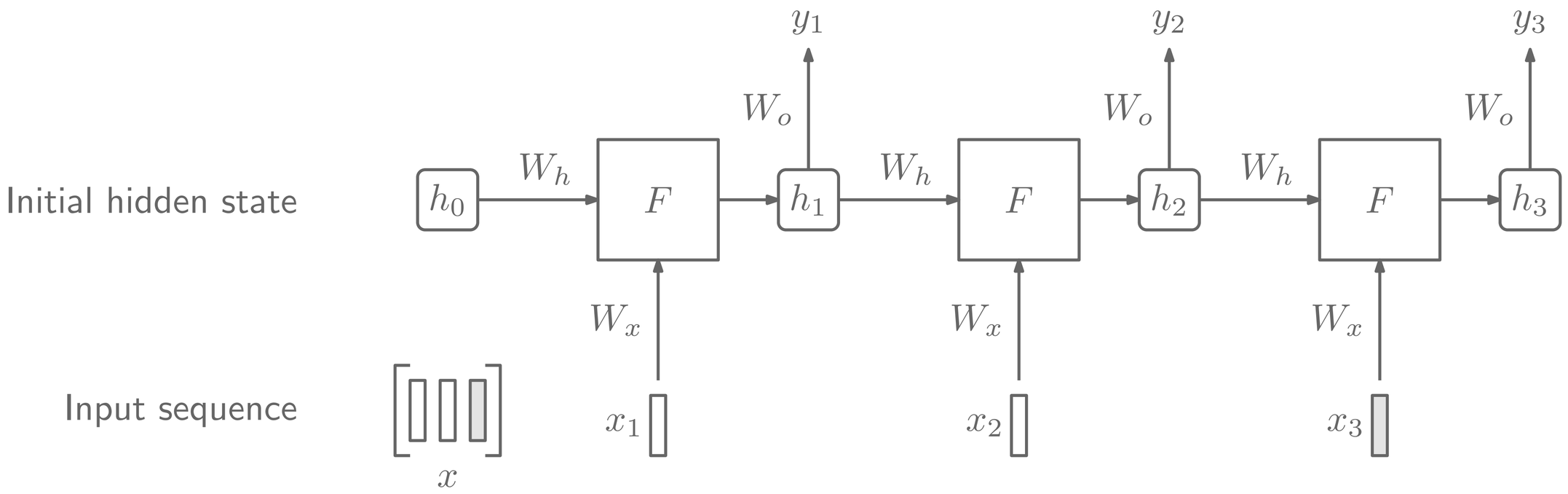
Input sequence



Unrolling time in RNNs

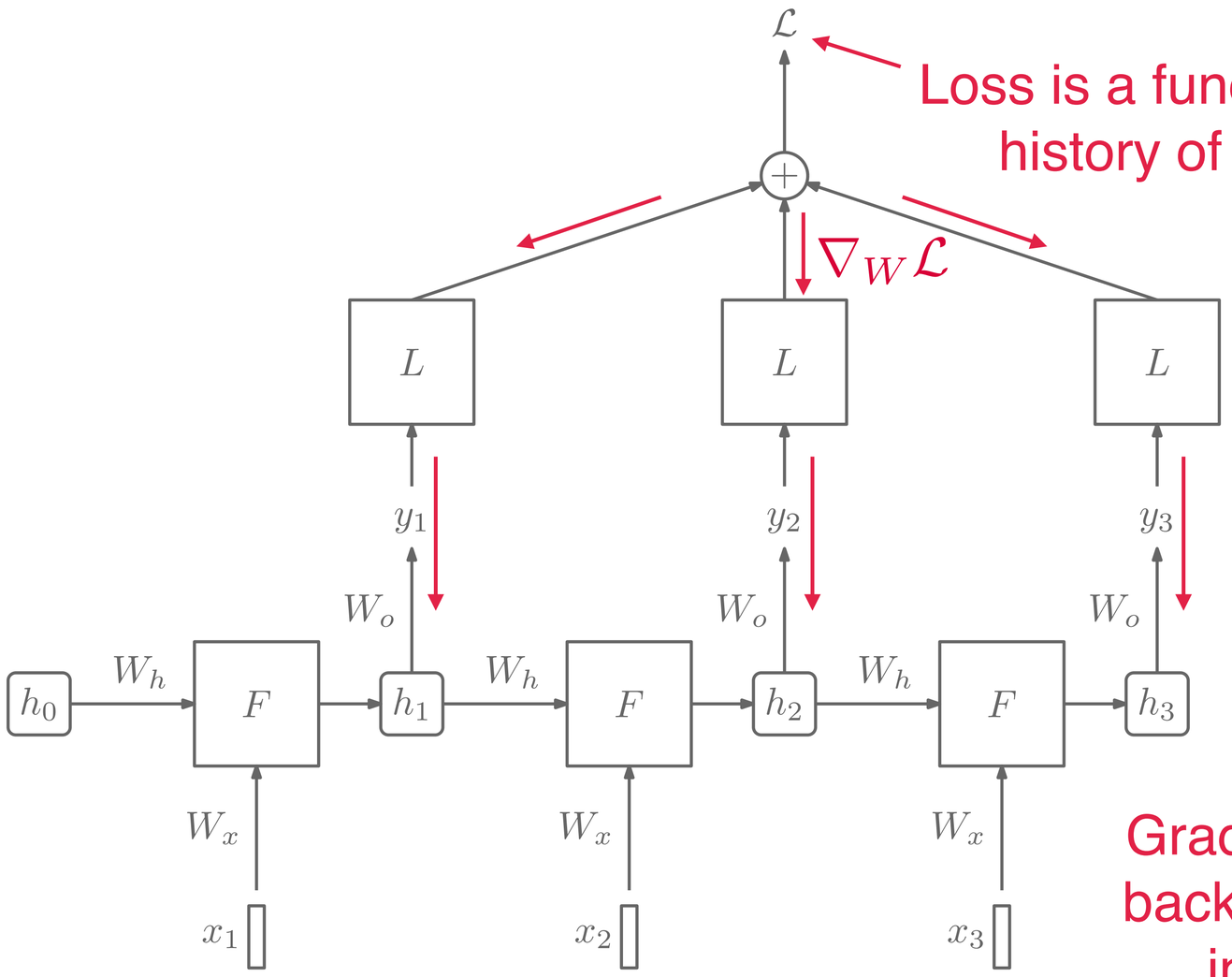


Unrolling time in RNNs



Unrolled graph as a variable network architecture

Unrolling time in RNNs

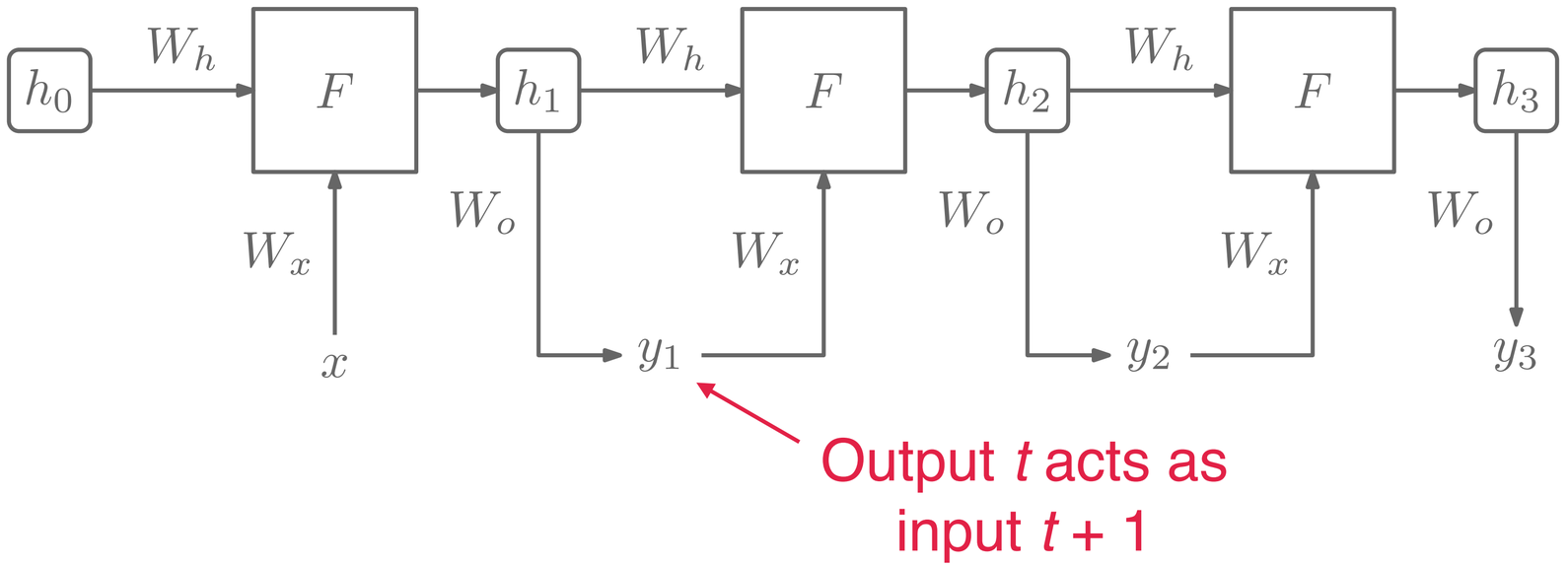


Backpropagation through time

- The unrolled graph is a well-formed computational graph
  - Autograd software can propagate the gradients back as in a standard graph
  - Parameters are shared across time
  - Derivatives aggregated across time
- The process is known as back-propagation through time

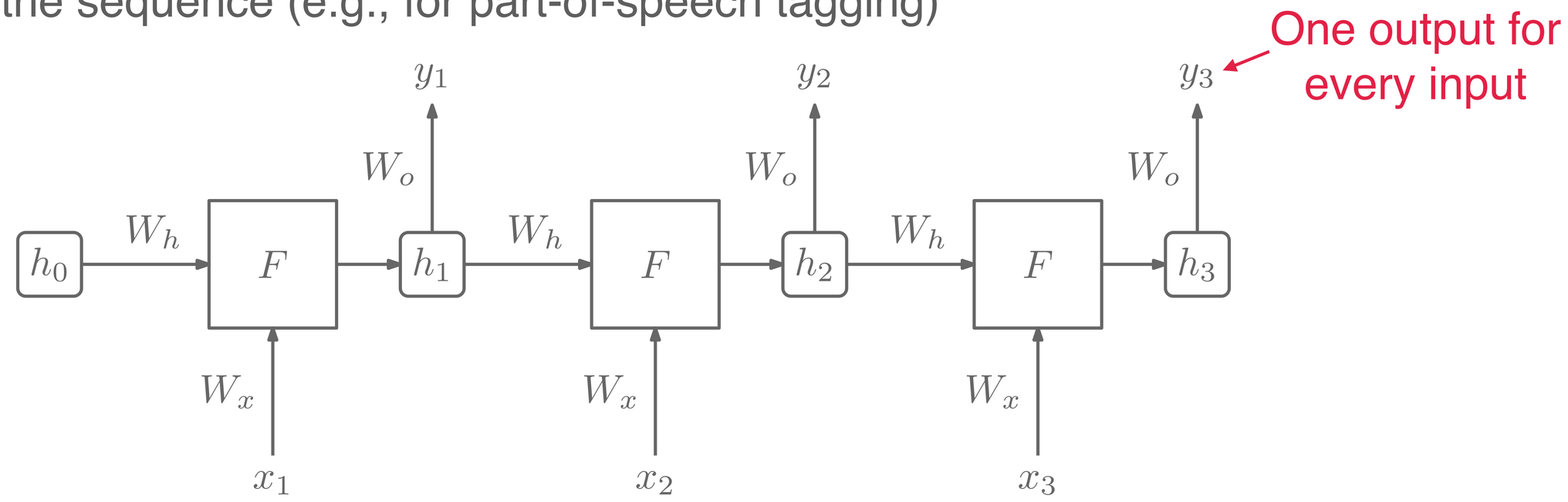
Applications of RNNs

- **Sequence generation:** Network generates a sequence of symbols as an auto-regressive model (e.g., for language generation)



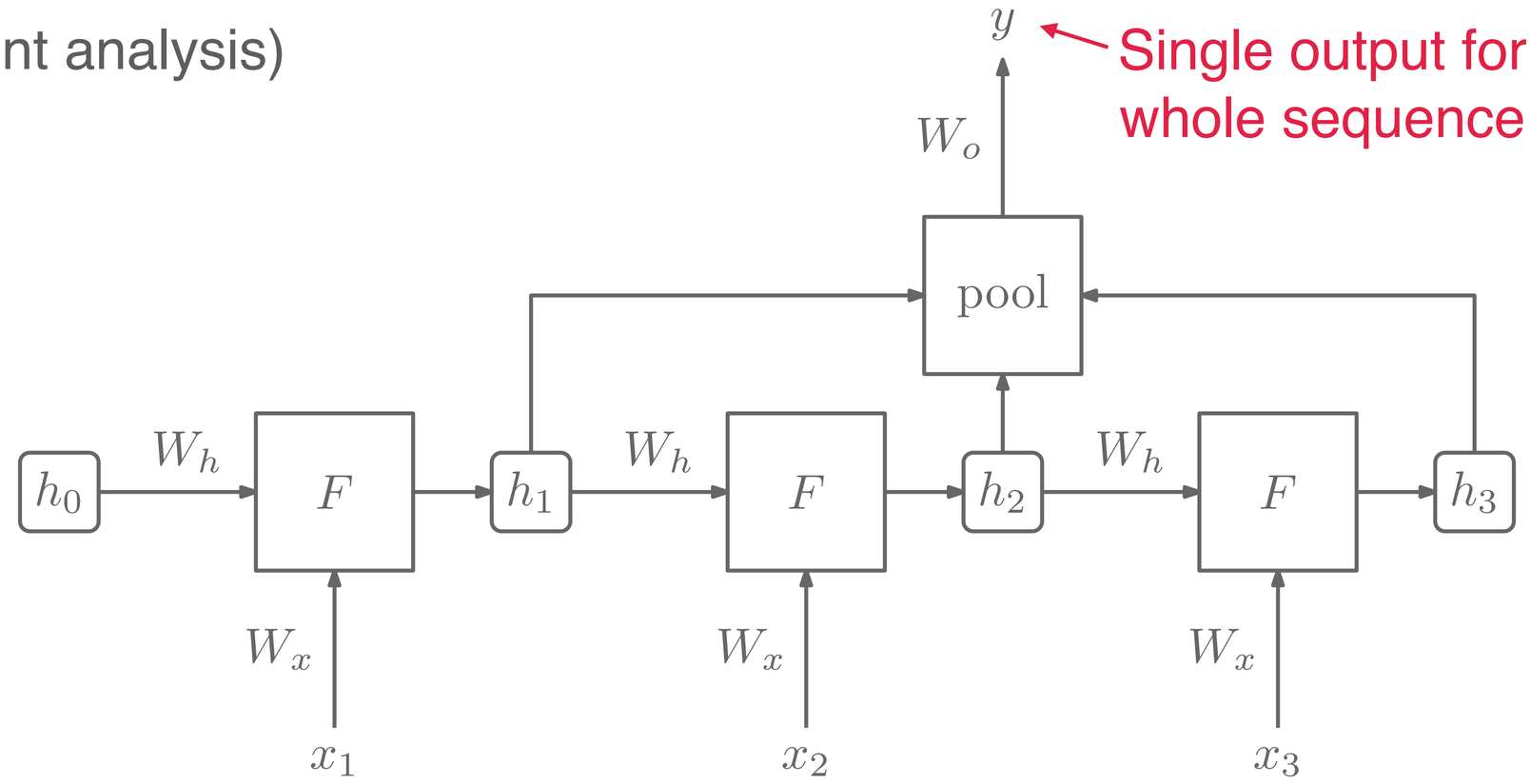
Applications of RNNs

- **Sequence tagging:** Takes a sequence as an input and returns a label/output for each element in the sequence (e.g., for part-of-speech tagging)



Applications of RNNs

- **Pooled classification:** Takes a sequence as an input and returns a single label (e.g., for sentiment analysis)



Sequence generation

Language modeling using RNNs

- Language models are used to represent/describe languages
- Most powerful models are probabilistic:

$$\mathbb{P}[\langle \mathsf{s} \rangle, x_1, x_2, \ldots, x_N, \langle \mathsf{e} \rangle] = \prod_{n=1}^{N+1} \mathbb{P}[x_n \mid x_0, \ldots, x_{n-1}]$$

Probability of a sentence

Probability of a word given previous words

- However, these distributions are too complex to learn

Sequence generation

Language modeling using RNNs

- **RNNs:**
  - Maintain a state vector h_t that is a function of previous state and current input:

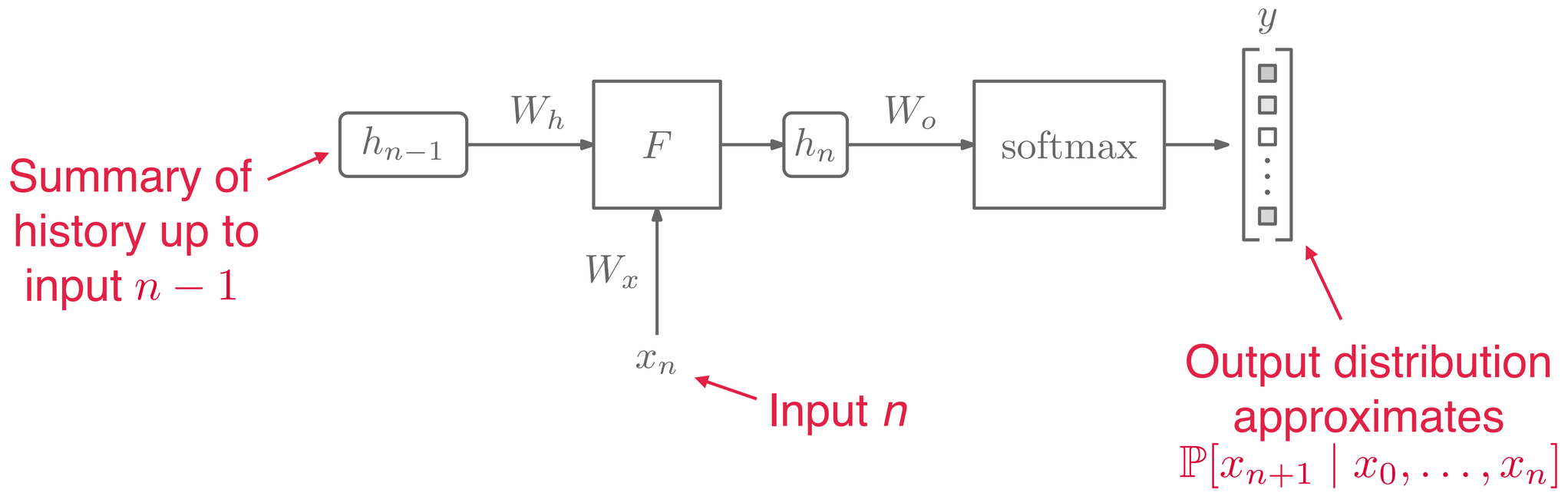
$$h_n = F(W_x x_n + W_h h_{n-1})$$

  - Compute probability of next element:

$$\mathbb{P}[x_{n+1} \mid x_0, \ldots, x_n] = \text{softmax}(W_o h_n)$$

Sequence generation

Language modeling using RNNs

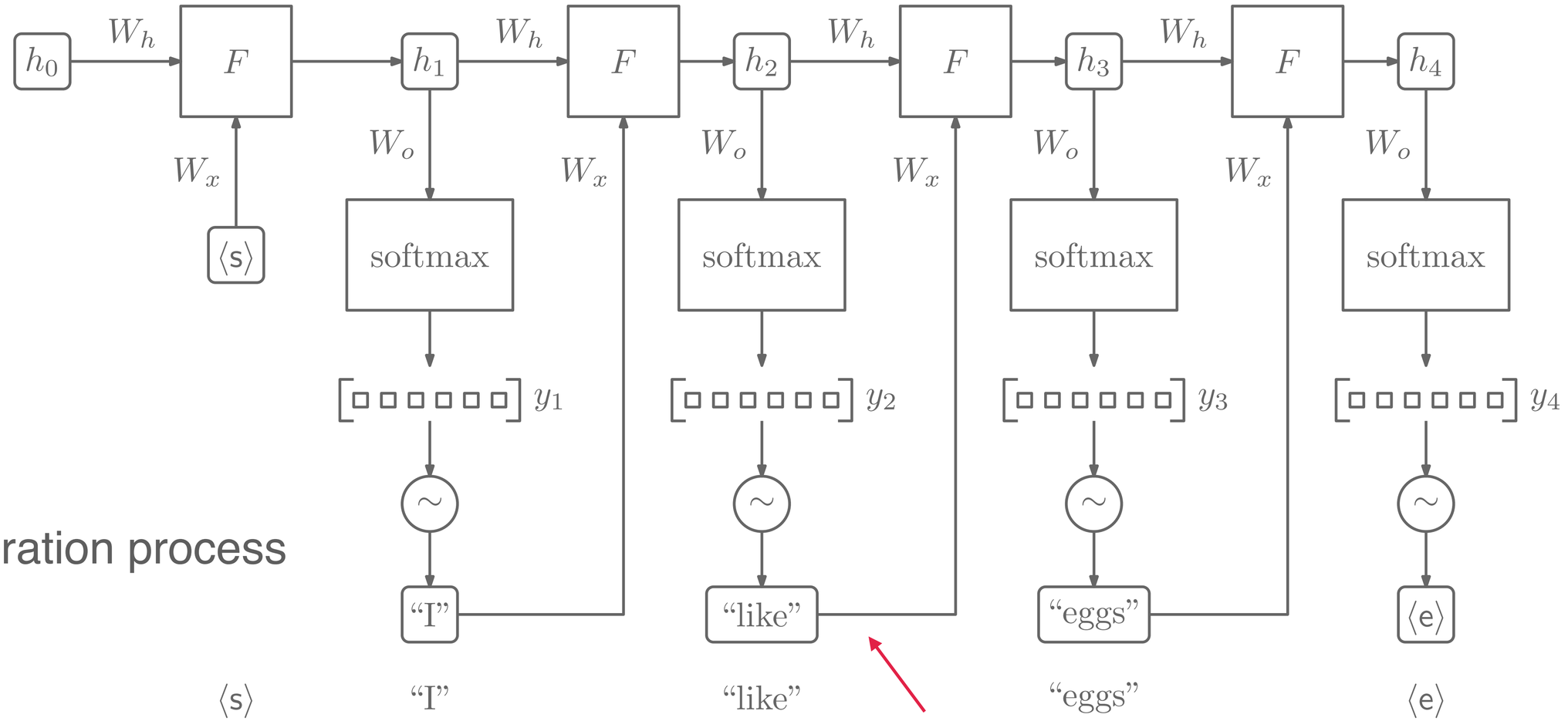


Generation process

- Compute $h_1 = F(W_x x_1 + W_h h_0)$, with $x_1 = \langle s \rangle$
- Sample $x_2 \sim \text{softmax}(W_o h_1)$
- Compute $h_2 = F(W_x x_2 + W_h h_1)$
- Sample $x_3 \sim \text{softmax}(W_o h_2)$
- … etc…

Sequence generation

Language modeling using RNNs



Generation process

Auto-regressive

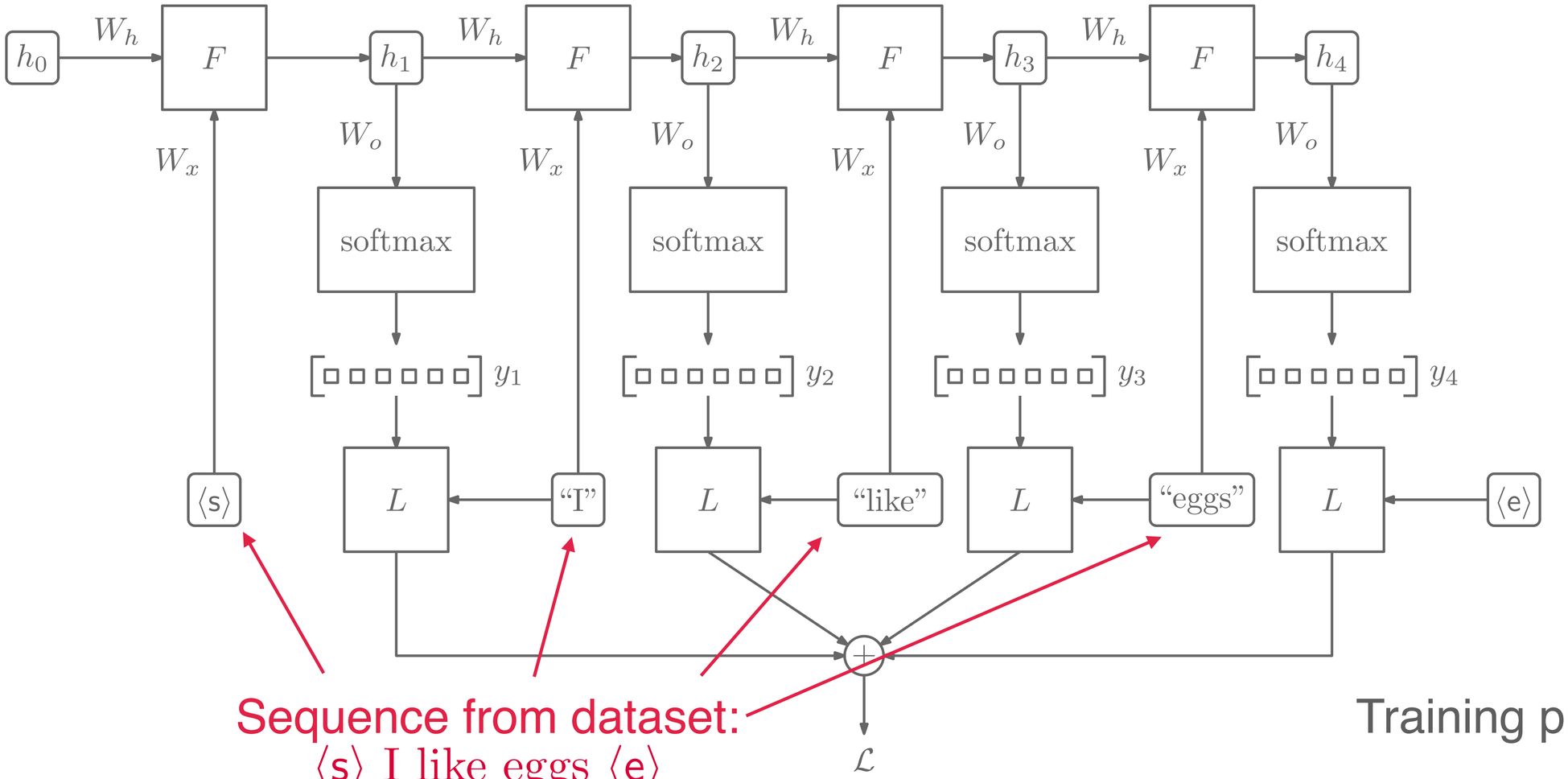
Sequence generation

Language modeling using RNNs

- The network is trained using maximum likelihood estimation
- We optimize the parameters to maximize the likelihood of the sequences/sentences in the dataset
- During training, we feed the network with elements from actual sequences in the dataset (teacher forcing)

Sequence generation

Language modeling using RNNs



Sequence generation

Language modeling using RNNs

- It is also possible to build a character-level language model
  - We will do a simple example in the lab
  - Advantage: Can generate any word, not just words from vocabulary
  - Disadvantage: Needs larger memory…

Sequence tagging

Part-of-speech tagging using RNNs

- Part-of-speech tagging consists of assigning a tag to words in a sentence:

Time	flies	like	an	arrow
noun	verb	prep	det	noun

- There is a high correlation between adjacent words

Sequence tagging

Part-of-speech tagging using RNNs

- **RNNs:**
  - Input is a sequence of word embeddings (x_1, …, x_N)
  - Network maintains a state vector

$$h_n = F(W_x x_n + W_h h_{n-1})$$

  - Compute probability of current element tag:

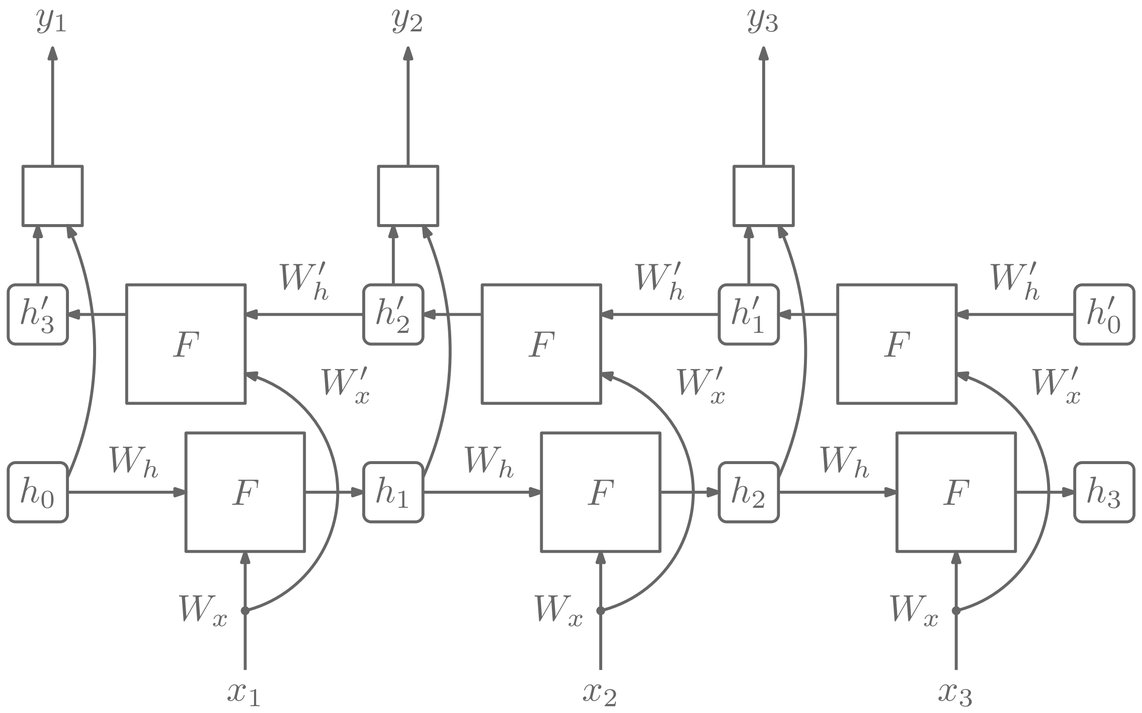
$$\mathbb{P}[c_n \mid x_0, \ldots, x_n] = \mathrm{softmax}(W_o h_n)$$

Tag of element *n*

Sequence tagging

Part-of-speech tagging using RNNs

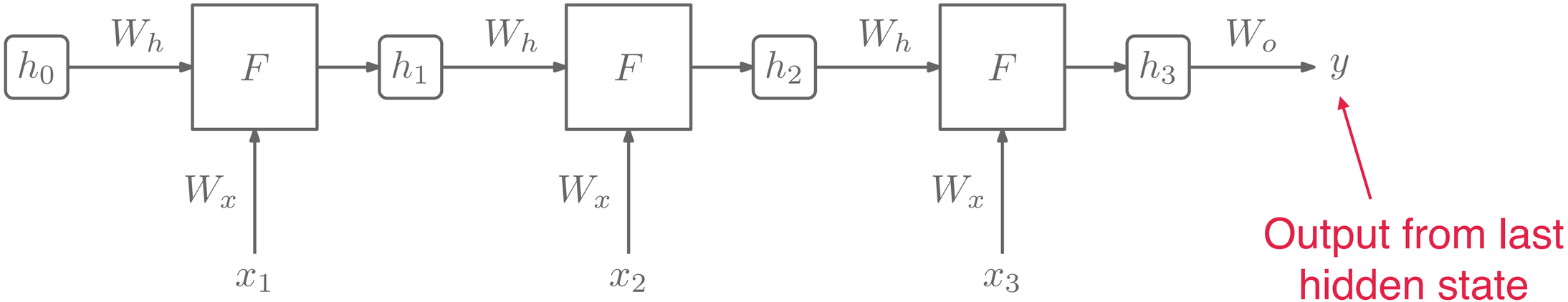
- The model can be improved:
  - The tag is conditioned only on adjacent words to the left
- We can use a bidirectional RNN to condition also on words to the right



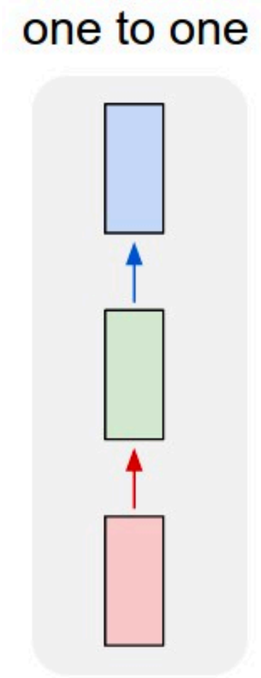
Pooled classification

- What if we want to predict a single label for the whole sequence (e.g., sentiment analysis)?
- We can still use an RNN to capture the information in the input sequence
- We pool the RNN states to a single output
  - Common strategy: use only the last network state

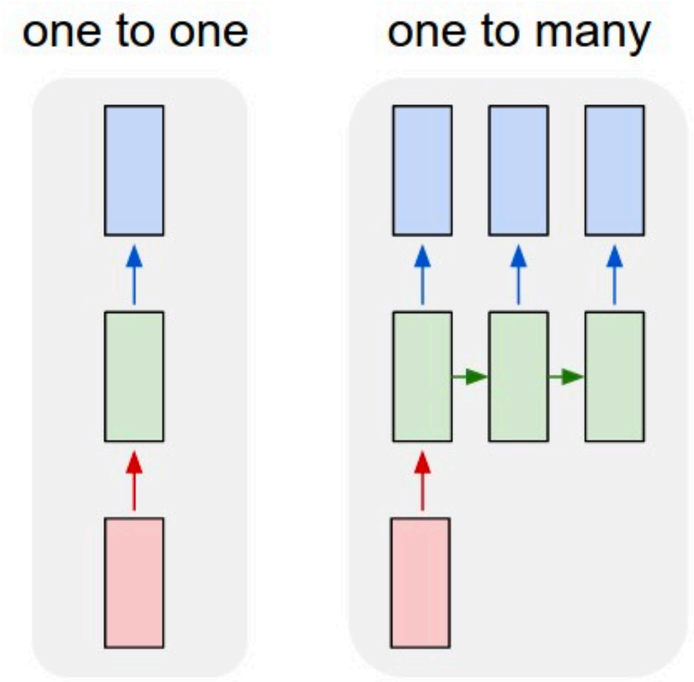
Pooled classification



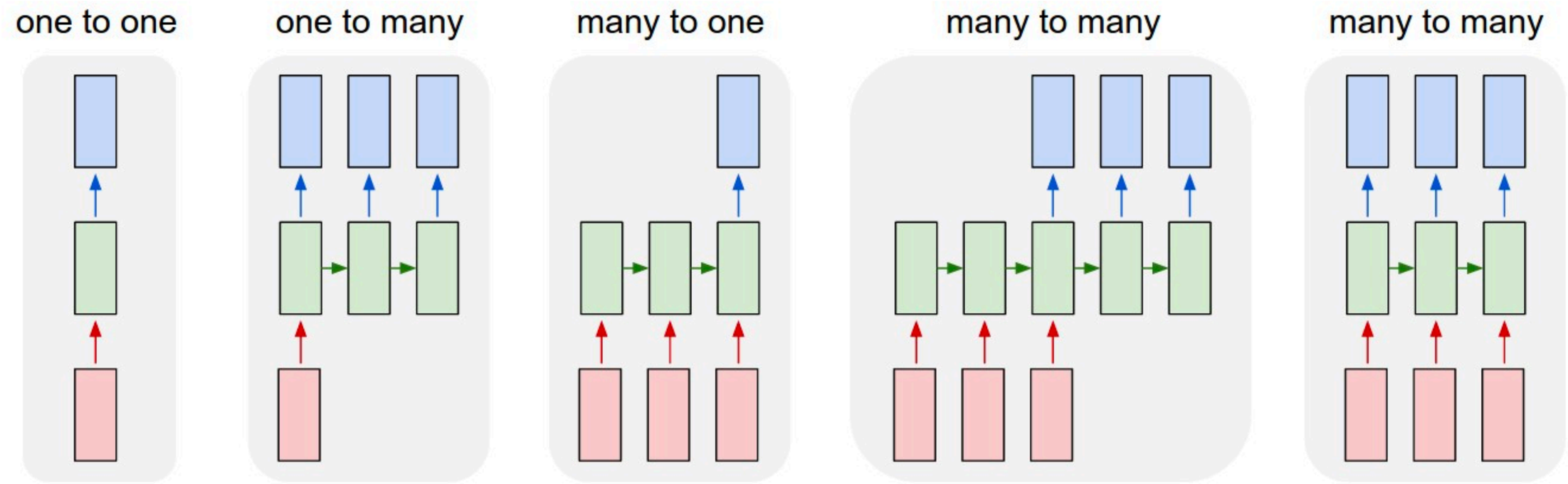
The versatility of RNNs



The versatility of RNNs

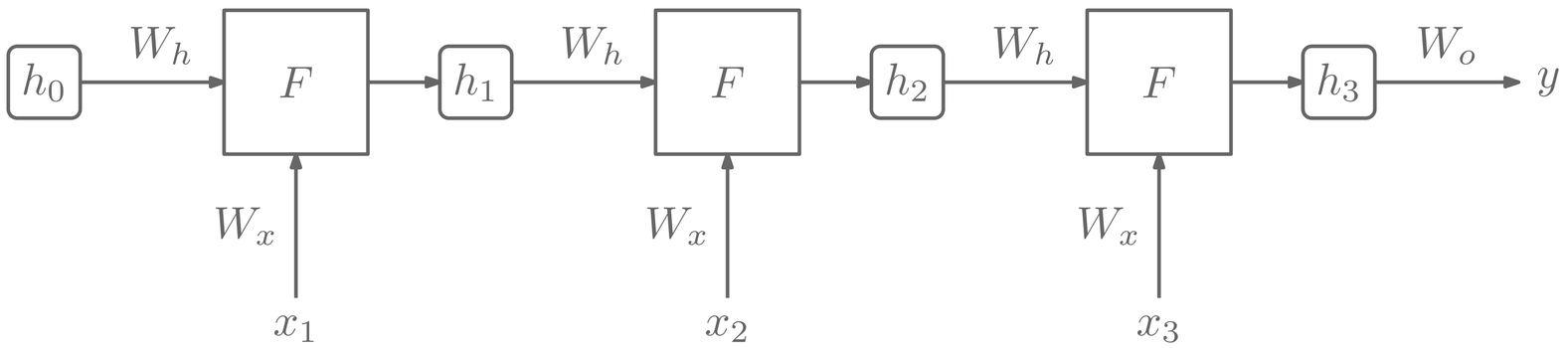


The versatility of RNNs



… however…

Exploding and vanishing gradients

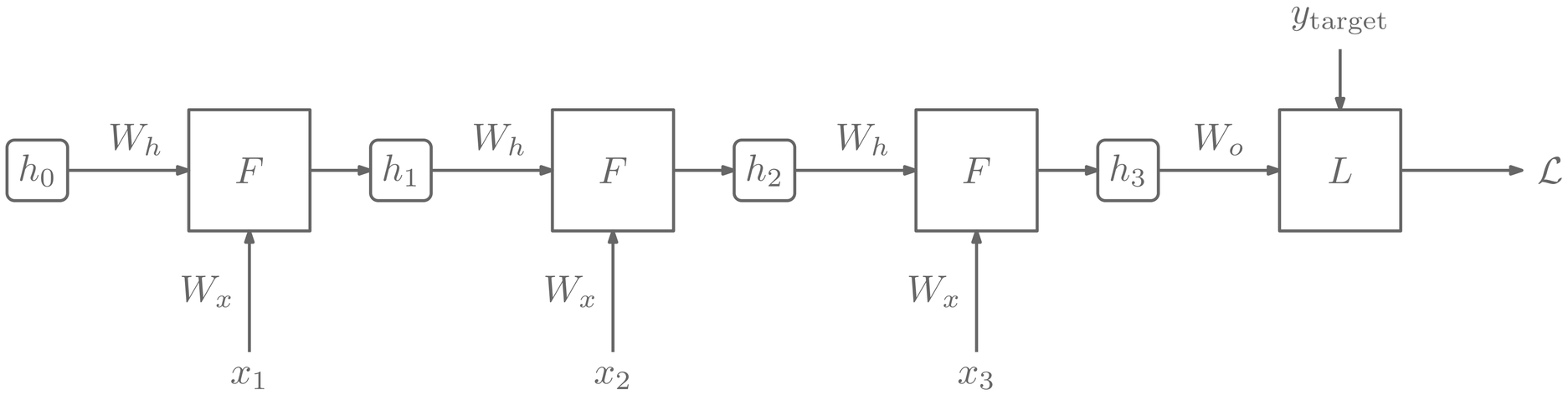


- We have:

$$h_t = F(W_x x_t + W_h h_{t-1})$$
$$y = W_o h_T$$

- What about the gradient?

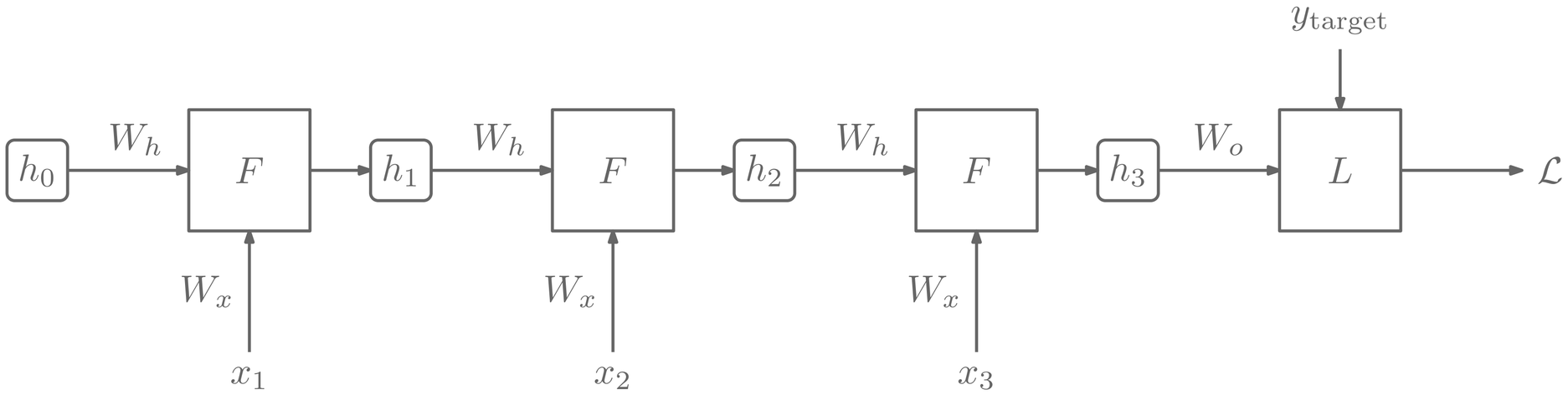
Exploding and vanishing gradients



- Computing the gradient we have, for example,

$$\frac{\partial \mathcal{L}}{\partial h_1} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h_T} \frac{\partial h_T}{\partial h_{T-1}} \cdots \frac{\partial h_2}{\partial h_1}$$

Exploding and vanishing gradients

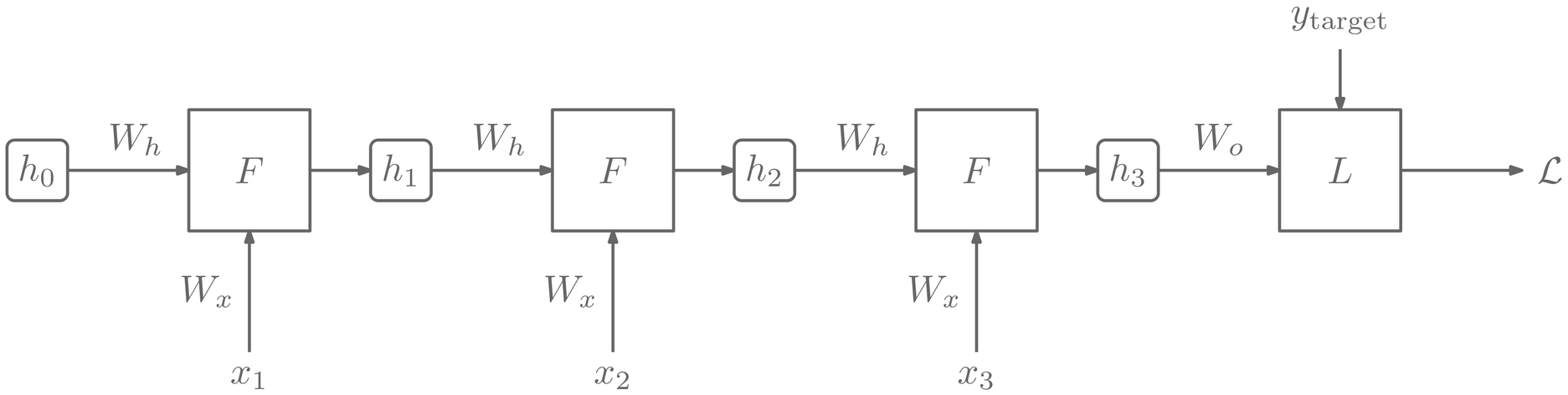


- Computing the gradient we have, for example,

$$\frac{\partial \mathcal{L}}{\partial h_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h_T}\prod_{t=2}^{T}\frac{\partial h_t}{\partial h_{t-1}}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diag}(F'(z_t))W_h$$

Exploding and vanishing gradients



- Computing the gradient we have, for example,

$$\frac{\partial \mathcal{L}}{\partial h_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h_T}\prod_{t=2}^{T}\text{diag}(F'(z_t))W_h$$

Same matrix multiplied *T* times

Exploding and vanishing gradients

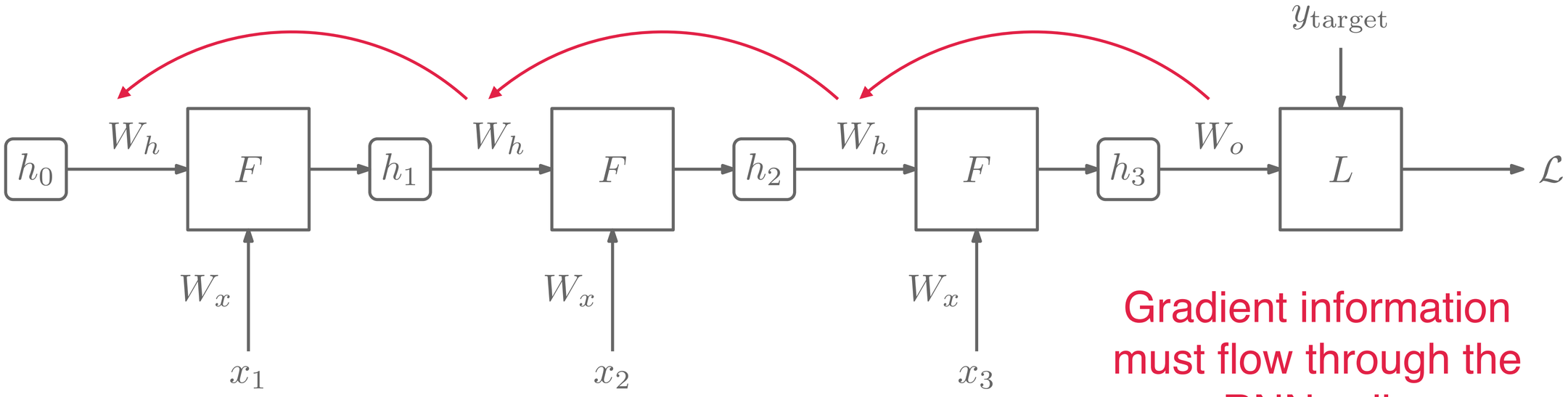
- Three cases:
  - If $\text{eig}(W_h) = 1$, gradient propagation is stable
  - If $\text{eig}(W_h) > 1$, gradient propagation explodes → Gradient clipping
  - If $\text{eig}(W_h) < 1$, gradient propagation vanishes
- Vanishing gradient is frequent and hard to deal with

Alternate RNN architectures

- Common approach to address vanishing gradients is to adopt alternative RNN architectures
  - Gated recurrent unit (GRU)
  - Long-short term memory (LSTM)

GRUs and LSTMs

The problem with the gradients

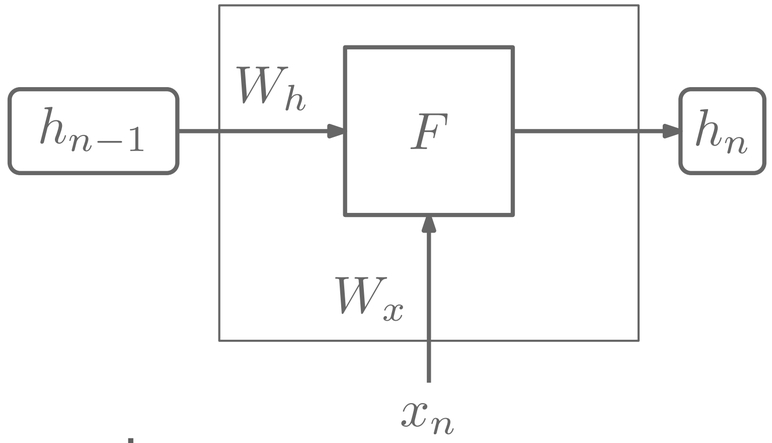


Gradient information must flow through the RNN cells

The problem with the gradients

- One possibility would be to add “short-circuits” to the network to facilitate the “flow” of gradient information

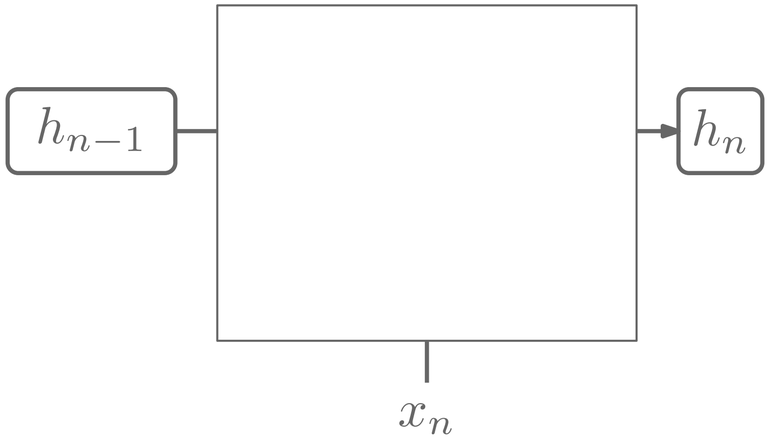
Recurrent Unit



- In a standard recurrent unit, we have:

$$h_n = F(W_x x_n + W_h h_{n-1})$$

Gated Recurrent Unit (GRU)

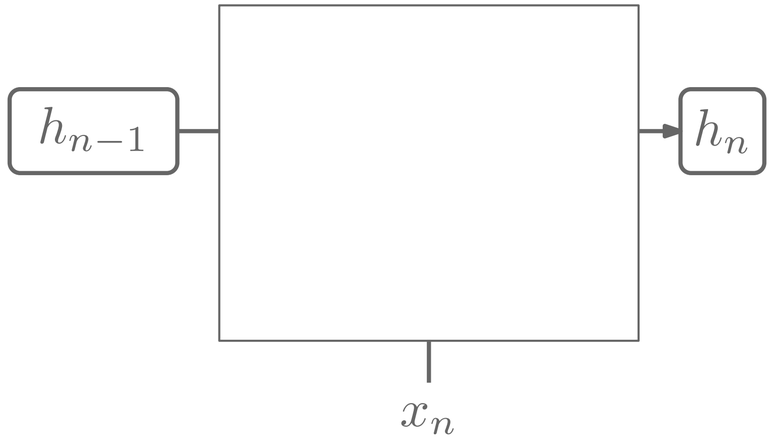


- Now…

$$h_n = F(W_x x_n + W_h h_{n-1})$$

Add a term that allows the unit to “forget”

Gated Recurrent Unit (GRU)



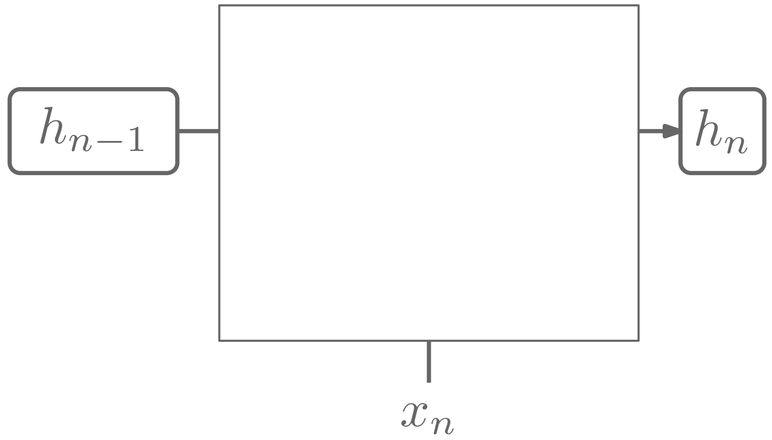
- Now…

$$\tilde{h}_n = F(W_x x_n + W_h(r_n \odot h_{n-1}))$$

Candidate internal state

Add a term that allows the unit to “forget”

Gated Recurrent Unit (GRU)



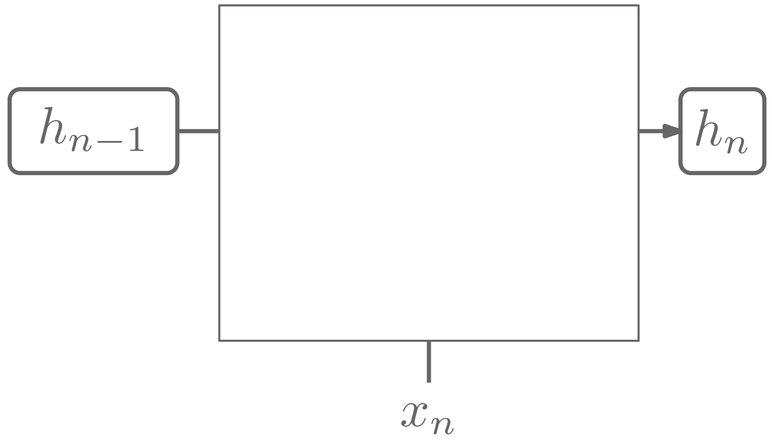
- Now…

$$\tilde{h}_n = F(W_x x_n + W_h(r_n \odot h_{n-1}))$$

$$h_n = u_n \odot \tilde{h}_n + (1 - u_n) \odot h_{n-1}$$

New internal state combines new candidate and old

Gated Recurrent Unit (GRU)

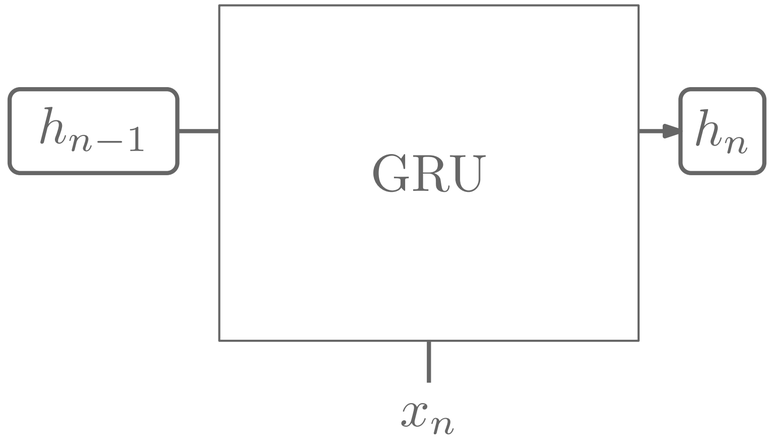


- Now…

$$\tilde{h}_n = F(W_x x_n + W_h(r_n \odot h_{n-1}))$$
$$h_n = u_n \odot \tilde{h}_n + (1 - u_n) \odot h_{n-1}$$
$$r_n = \sigma(W_{x,r} x_n + W_{h,r} h_{n-1})$$

Reset signal computed from current input and previous state

Gated Recurrent Unit (GRU)



- Now…

$$\tilde{h}_n = F(W_x x_n + W_h(r_n \odot h_{n-1}))$$

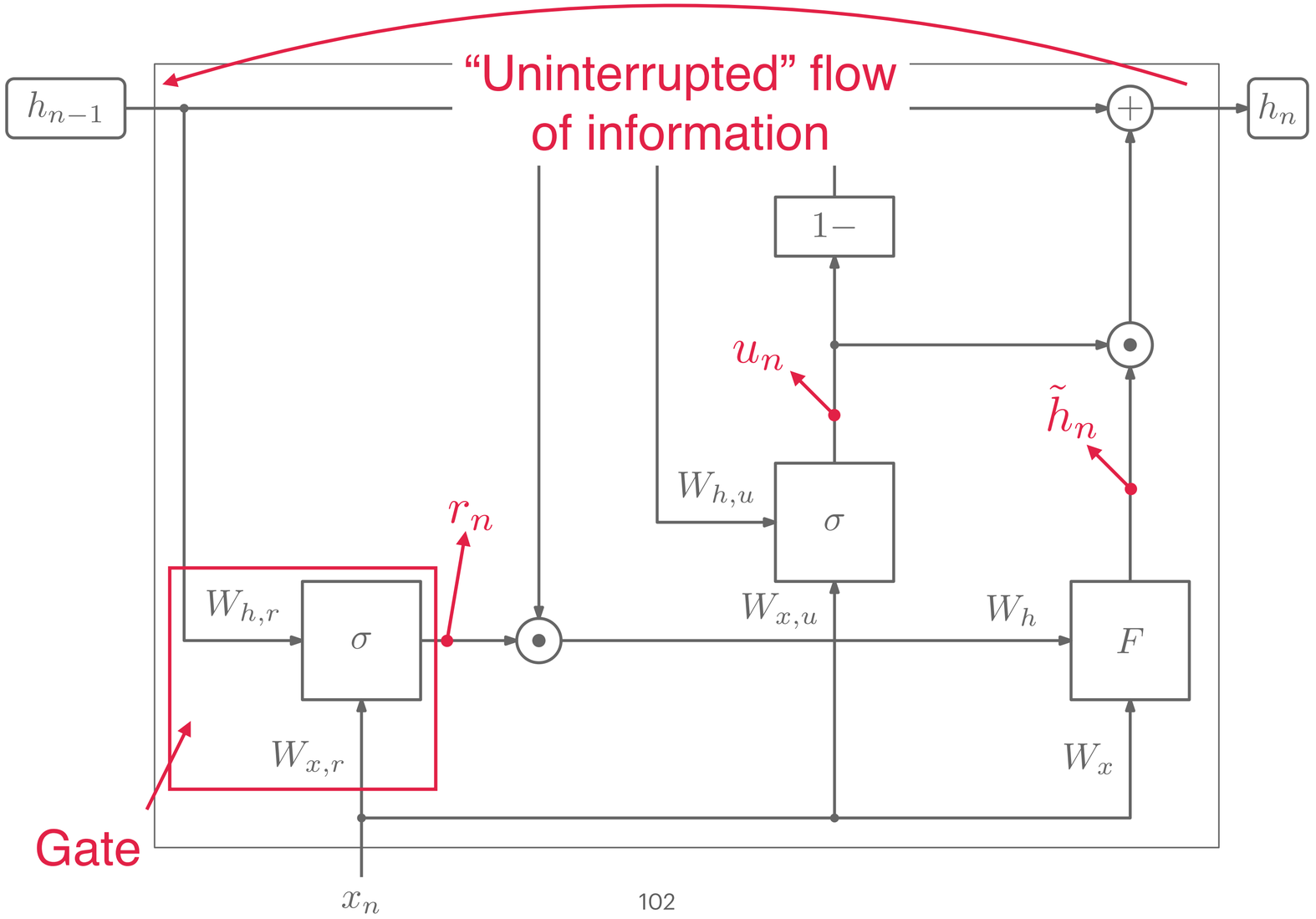
$$h_n = u_n \odot \tilde{h}_n + (1 - u_n) \odot h_{n-1}$$

$$r_n = \sigma(W_{x,r} x_n + W_{h,r} h_{n-1})$$

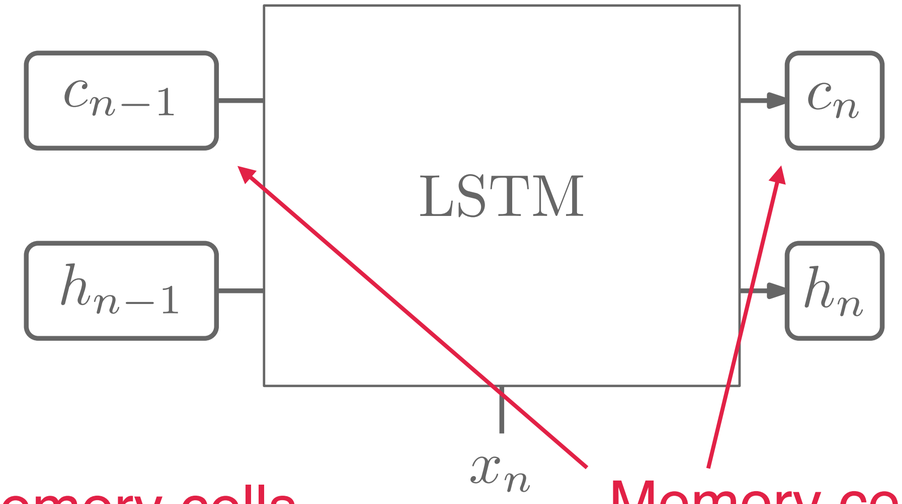
$$u_n = \sigma(W_{x,u} x_n + W_{h,u} h_{n-1})$$

Update signal computed from current input and previous state

Gated Recurrent Unit (GRU)



Long-Short Term Memory (LSTM)



- LSTM units also include memory cells
  - A second internal vector that is updated additively
  - Memory cells controlled through input, forget and output gates

Long-Short Term Memory (LSTM)

- In an LSTM,

$$\tilde{c}_n = F_c(W_{x,c}x_n + W_{h,c}h_{n-1})$$

Candidate
cell value

Long-Short Term Memory (LSTM)

- In an LSTM,

$$\tilde{c}_n = F_c(W_{x,c}x_n + W_{h,c}h_{n-1})$$
$$c_n = i_n \odot \tilde{c}_n + f_n \odot c_{n-1}$$

New cell value

Input gate (controls input of new information)

Forget gate (controls forgetting of old information)

Long-Short Term Memory (LSTM)

- In an LSTM,

$$\tilde{c}_n = F_c(W_{x,c}x_n + W_{h,c}h_{n-1})$$
$$c_n = i_n \odot \tilde{c}_n + f_n \odot c_{n-1}$$
$$h_n = o_n \odot F_h(c_n)$$

Internal state

Output gate (controls how much memory info goes out)

Long-Short Term Memory (LSTM)

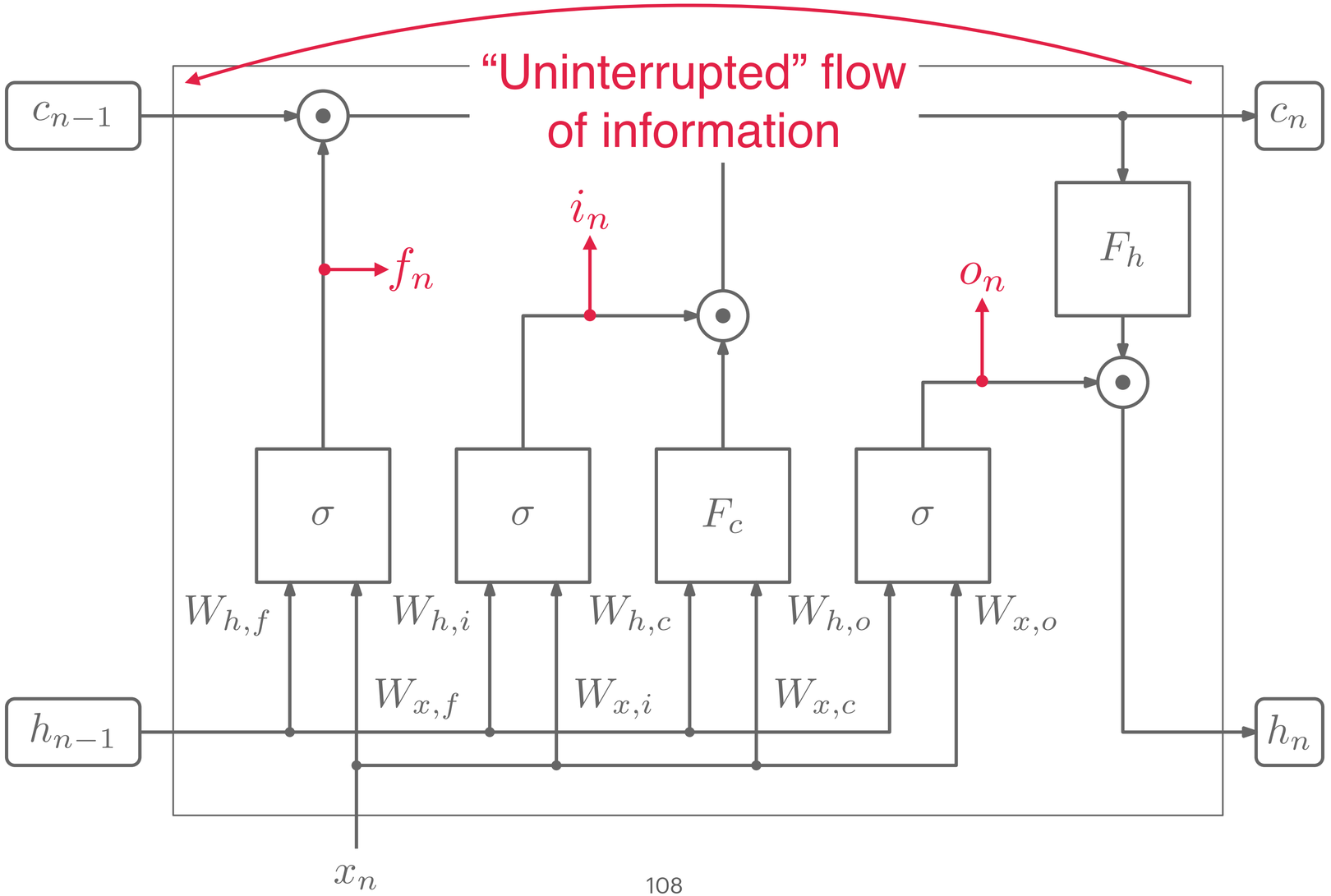
- In an LSTM,

$$\tilde{c}_n = F_c(W_{x,c}x_n + W_{h,c}h_{n-1})$$
$$c_n = i_n \odot \tilde{c}_n + f_n \odot c_{n-1}$$
$$h_n = o_n \odot F_h(c_n)$$

$$i_n = \sigma(W_{x,i}x_n + W_{h,i}h_{n-1})$$
$$f_n = \sigma(W_{x,f}x_n + W_{h,f}h_{n-1})$$
$$o_n = \sigma(W_{x,o}x_n + W_{h,o}h_{n-1})$$

Gates

Long-Short Term Memory (LSTM)

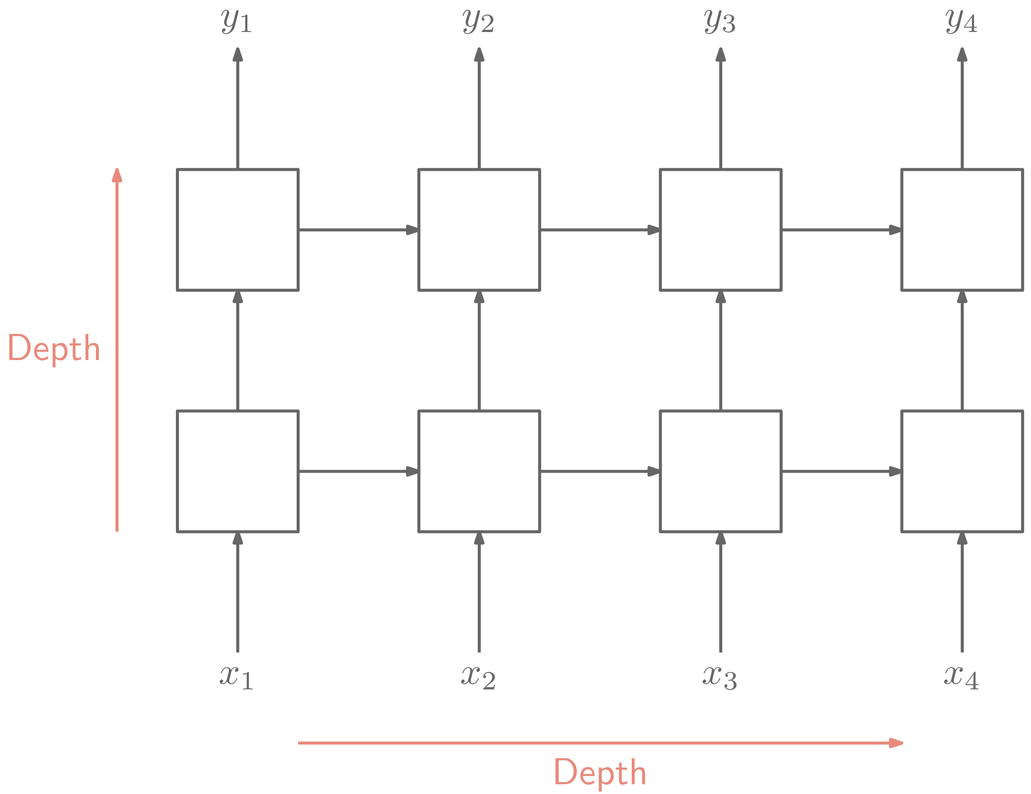


Final remarks

- LSTMs have better gradient propagation thanks to additive updates
- Variants exist that simplify some of the gates
- Have been used extensively in NLP tasks:
  - Named entity recognition
  - Machine translation
  - …

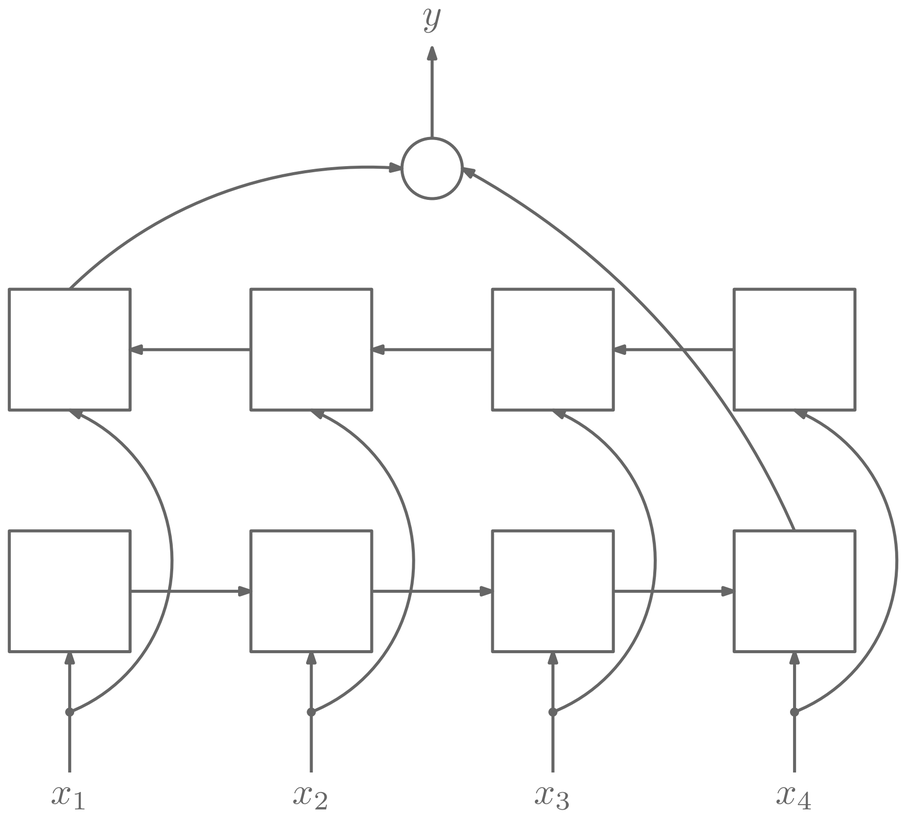
Final remarks

- Depth in recurrent layers helps (standard 2-8 recurrent layers)



Final remarks

- Bidirectional RNNs can be implemented with standard RNN cells, or GRUs or LSTMs



Final remarks

- Dropout can be added between layers, but not on recurrent connections
- For mini-batching, it is often necessary to pad the sequences to ensure constant length within the mini-batch

Summary

- Neural nets learn internal representations that can be transferred across tasks
- Distributed representations are exponentially more compact and allow generalizing to unseen objects
- Auto-encoders are an effective means for learning representations
- Word embeddings are continuous representations of words that are extremely useful in NLP

Summary

- Recurrent neural networks allow to take advantage of sequential input structure
- They can generate, tag, and classify sequences
- RNNs are trained with back-propagation through time
- Vanilla RNNs suffer from vanishing and exploding gradients
- LSTMs and other gated units are more complex RNNs that avoid vanishing gradients
- They can be extended to other structures like trees, images, and graphs

References

- Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. “A neural probabilistic language model.” *Journal of Machine Learning Research*, 3:1137–1155, 2003.
- Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. “Learning phrase representations using RNN encoder-decoder for statistical machine translation.” In *Proc. Empirical Methods in Natural Language Processing*, 2014.
- Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. “Why does unsupervised pre-training help deep learning?” *Journal of Machine Learning Research*, 11:625–660, 2010.
- Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016.
- Hochreiter, S. and Schmidhuber, J. “Long short-term memory.” *Neural computation*, 9(8):1735–1780, 1997.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. “Distributed representations of words and phrases and their compositionality.” In *Adv. Neural Information Processing Systems*, pages 3111–3119, 2013.